W3School C语言教程

wizardforcel

Published with GitBook



目錄

W3School C语言教程

来源：C语言教程

整理：飞龙

C语言教程

C 简介

C 语言是一种通用的高级语言，最初是由丹尼斯·里奇在贝尔实验室为开发 UNIX 操作系统而设计的。C 语言最开始是于 1972 年在 DEC PDP-11 计算机上被首次实现。

在 1978 年，布莱恩·柯林汉（Brian Kernighan）和丹尼斯·里奇（Dennis Ritchie）制作了 C 的第一个公开可用的描述，现在被称为 K&R 标准。

UNIX 操作系统，C编译器，和几乎所有的 UNIX 应用程序都是用 C 语言编写的。由于各种原因，C 语言现在已经成为一种广泛使用的专业语言。

- 易于学习。
- 结构化语言。
- 它产生高效率的程序。
- 它可以处理底层的活动。
- 它可以在多种计算机平台上编译。

关于 C

- C 语言是为了编写 UNIX 操作系统而被发明的。
- C 语言是以 B 语言为基础的，B 语言大概是在 1970 年被引进的。
- C 语言标准是于 1988 年由美国国家标准协会（ANSI，全称 American National Standard Institute）制定的。
- 截至 1973 年，UNIX 操作系统完全使用 C 语言编写。
- 目前，C 语言是最广泛使用的系统程序设计语言。
- 大多数先进的软件都是使用 C 语言实现的。
- 当今最流行的 Linux 操作系统和 RBDMS MySQL 都是使用 C 语言编写的。

为什么要使用 C？

C 语言最初是用于系统开发工作，特别是组成操作系统的程序。由于 C 语言所产生的代码运行速度与汇编语言编写的代码运行速度几乎一样，所以采用 C 语言作为系统开发语言。下面列举几个使用 C 的实例：

- 操作系统
- 语言编译器
- 汇编器
- 文本编辑器
- 打印假脱机

- 网络驱动器
- 现代程序
- 数据库
- 语言解释器
- 实体工具

C 程序

一个 C 语言程序，可以是 3 行，也可以是数百万行，它可以写在一个或多个扩展名为 **".c"** 的文本文件中，例如，*hello.c*。您可以使用 **"vi"**、**"vim"** 或任何其他文本编辑器来编写您的 C 语言程序。

本教程假定您已经知道如何编辑一个文本文件，以及如何在程序文件中编写源代码。

C 环境设置

本地环境设置

如果您想要设置 C 语言环境，您需要确保电脑上有以下两款可用的软件，文本编辑器和 C 编译器。

文本编辑器

这将用于输入您的程序。文本编辑器包括 Windows Notepad、OS Edit command、Brief、Epsilon、EMACS 和 vim/vi。

文本编辑器的名称和版本在不同的操作系统上可能会有所不同。例如，Notepad 通常用于 Windows 操作系统上，vim/vi 可用于 Windows 和 Linux/UNIX 操作系统上。

通过编辑器创建的文件通常称为源文件，源文件包含程序源代码。C 程序的源文件通常使用扩展名 "**.c**"。

在开始编程之前，请确保您有一个文本编辑器，且有足够的经验来编写一个计算机程序，然后把它保存在一个文件中，编译并执行它。

C 编译器

写在源文件中的源代码是人类可读的源。它需要"编译"，转为机器语言，这样 CPU 可以按给定指令执行程序。

C 语言编译器用于把源代码编译成最终的可执行程序。这里假设您已经对编程语言编译器有基本的了解了。

最常用的免费可用的编译器是 GNU 的 C/C++ 编译器，如果您使用的是 HP 或 Solaris，则可以使用各自操作系统上的编译器。

以下部分将指导您如何在不同的操作系统上安装 GNU 的 C/C++ 编译器。这里同时提到 C/C++，主要是因为 GNU 的 gcc 编译器适合于 C 和 C++ 编程语言。

UNIX/Linux 上的安装

如果您使用的是 **Linux** 或 **UNIX**，请在命令行使用下面的命令来检查您的系统上是否安装了 GCC：

```
$ gcc -v
```

如果您的计算机上已经安装了 GNU 编译器，则会显示如下消息：

```
Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix=/usr .......
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

如果未安装 GCC，那么您必须按照 http://gcc.gnu.org/install/ 上的详细说明安装 GCC。

本教程是基于 Linux 编写的，所有给定的实例都已在 Cent OS Linux 系统上编译过。

Mac OS 上的安装

如果您使用的是 Mac OS X，最快捷的获取 GCC 的方法是从苹果的网站上下载 Xcode 开发环境，并按照安装说明进行安装。一旦安装上 Xcode，您就能使用 GNU 编译器。

Xcode 目前可从 developer.apple.com/technologies/tools/ 上下载。

Windows 上的安装

为了在 Windows 上安装 GCC，您需要安装 MinGW。为了安装 MinGW，请访问 MinGW 的主页 www.mingw.org，进入 MinGW 下载页面，下载最新版本的 MinGW 安装程序，命名格式为 MinGW-<version>.exe。

当安装 MinWG 时，您至少要安装 gcc-core、gcc-g++、binutils 和 MinGW runtime，但是一般情况下都会安装更多其他的项。

添加您安装的 MinGW 的 bin 子目录到您的 **PATH** 环境变量中，这样您就可以在命令行中通过简单的名称来指定这些工具。

当完成安装时，您可以从 Windows 命令行上运行 gcc、g++、ar、ranlib、dlltool 和其他一些 GNU 工具。

C 程序结构

在我们学习 C 语言的基本构建块之前，让我们先来看看一个最小的 C 程序结构，在接下来的章节中可以以此作为参考。

C Hello World 实例

C 程序主要包括以下部分：

- 预处理器指令
- 函数
- 变量
- 语句 & 表达式
- 注释

让我们看一段简单的代码，可以输出单词 "Hello World"：

```
#include <stdio.h>

int main()
{
   /* 我的第一个 C 程序 */
   printf("Hello, World! \n");

   return 0;
}
```

接下来我们讲解一下上面这段程序：

1. 程序的第一行 *#include <stdio.h>* 是预处理器指令，告诉 C 编译器在实际编译之前要包含 stdio.h 文件。
2. 下一行 *int main()* 是主函数，程序从这里开始执行。
3. 下一行 /.../ 将会被编译器忽略，这里放置程序的注释内容。它们被称为程序的注释。
4. 下一行 *printf(...)* 是 C 中另一个可用的函数，会在屏幕上显示消息 "Hello, World!"。
5. 下一行 **return 0;** 终止 main() 函数，并返回值 0。

编译 & 执行 C 程序

接下来让我们看看如何把源代码保存在一个文件中，以及如何编译并运行它。下面是简单的步骤：

1. 打开一个文本编辑器，添加上述代码。
2. 保存文件为 *hello.c*。

3. 打开命令提示符，进入到保存文件所在的目录。
4. 键入 *gcc hello.c*，输入回车，编译代码。
5. 如果代码中没有错误，命令提示符会跳到下一行，并生成 *a.out* 可执行文件。
6. 现在，键入 *a.out* 来执行程序。
7. 您可以看到屏幕上显示 *"Hello World"*。

```
$ gcc hello.c
$ ./a.out
Hello, World!
```

请确保您的路径中已包含 gcc 编译器，并确保在包含源文件 hello.c 的目录中运行它。

C 基本语法

我们已经看过 C 程序的基本结构，这将有助于我们理解 C 语言的其他基本的构建块。

C 的令牌（Tokens）

C 程序由各种令牌组成，令牌可以是关键字、标识符、常量、字符串值，或者是一个符号。例如，下面的 C 语句包括五个令牌：

```
printf("Hello, World! \n");
```

这五个令牌分别是：

```
printf
(
"Hello, World! \n"
)
;
```

分号；

在 C 程序中，分号是语句结束符。也就是说，每个语句必须以分号结束。它表明一个逻辑实体的结束。

例如，下面是两个不同的语句：

```
printf("Hello, World! \n");
return 0;
```

注释

注释就像是 C 程序中的帮助文本，它们会被编译器忽略。它们以 / 开始，以字符 / 终止，如下所示：

```
/* 我的第一个 C 程序 */
```

您不能在注释内嵌套注释，注释也不能出现在字符串或字符值中。

标识符

C 标识符是用来标识变量、函数，或任何其他用户自定义项目的名称。一个标识符以字母 A-Z 或 a-z 或下划线 _ 开始，后跟零个或多个字母、下划线和数字（0-9）。

C 标识符内不允许出现标点字符，比如 @、$ 和 %。C 是区分大小写的编程语言。因此，在 C 中，*Manpower* 和 *manpower* 是两个不同的标识符。下面列出几个有效的标识符：

```
mohd       zara    abc   move_name  a_123
myname50   _temp   j     a23b9      retVal
```

关键字

下表列出了 C 中的保留字。这些保留字不能作为常量名、变量名或其他标识符名称。

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

C 中的空格

只包含空格的行，可能带有注释，被称为空白行，C 编译器会完全忽略它。

在 C 中，空格用于描述空白符、制表符、换行符和注释。空格分隔语句的各个部分，让编译器能识别语句中的某个元素（比如 int）在哪里结束，下一个元素在哪里开始。因此，在下面的语句中：

```
int age;
```

在这里，int 和 age 之间必须至少有一个空格字符（通常是一个空白符），这样编译器才能够区分它们。另一方面，在下面的语句中：

```
fruit = apples + oranges;   // 获取水果的总数
```

fruit 和 =，或者 = 和 apples 之间没有必要有空格字符，但是为了增强可读性，您可以根据需要适当增加一些空格。

C 数据类型

在 C 语言中，数据类型指的是用于声明不同类型的变量或函数的一个广泛的系统。变量的类型决定了变量存储占用的空间，以及如何解释存储的位模式。

C 中的类型可分为以下几种：

类型	描述
基本类型：	它们是算术类型，包括两种类型：整数类型和浮点类型。
枚举类型：	它们也是算术类型，被用来定义在程序中只能赋予其一定的离散整数值的变量。
void 类型：	类型说明符 *void* 表明没有可用的值。
派生类型：	它们包括：指针类型、数组类型、结构类型、共用体类型和函数类型。

数组类型和结构类型统称为聚合类型。函数的类型指的是函数返回值的类型。在本章节接下来的部分我们将介绍基本类型，其他几种类型会在后边几个章节中进行讲解。

整数类型

下表列出了关于标准整数类型的存储大小和值范围的细节：

类型	存储大小	值范围
char	1 byte	-128 到 127 或 0 到 255
unsigned char	1 byte	0 到 255
signed char	1 byte	-128 到 127
int	2 或 4 bytes	-32,768 到 32,767 或 -2,147,483,648 到 2,147,483,647
unsigned int	2 或 4 bytes	0 到 65,535 或 0 到 4,294,967,295
short	2 bytes	-32,768 到 32,767
unsigned short	2 bytes	0 到 65,535
long	4 bytes	-2,147,483,648 到 2,147,483,647
unsigned long	4 bytes	0 到 4,294,967,295

为了得到某个类型或某个变量在特定平台上的准确大小，您可以使用 **sizeof** 运算符。表达式 *sizeof(type)* 得到对象或类型的存储字节大小。下面的实例演示了获取 int 类型的大小：

```
#include <stdio.h>
#include <limits.h>

int main()
{
   printf("Storage size for int : %d \n", sizeof(int));

   return 0;
}
```

当您在 Linux 上编译并执行上面的程序时，它会产生下列结果：

```
Storage size for int : 4
```

浮点类型

下表列出了关于标准浮点类型的存储大小、值范围和精度的细节：

类型	存储大小	值范围	精度
float	4 byte	1.2E-38 到 3.4E+38	6 位小数
double	8 byte	2.3E-308 到 1.7E+308	15 位小数
long double	10 byte	3.4E-4932 到 1.1E+4932	19 位小数

头文件 float.h 定义了宏，在程序中可以使用这些值和其他有关实数二进制表示的细节。下面的实例将输出浮点类型占用的存储空间以及它的范围值：

```
#include <stdio.h>
#include <float.h>

int main()
{
   printf("Storage size for float : %d \n", sizeof(float));
   printf("Minimum float positive value: %E\n", FLT_MIN );
   printf("Maximum float positive value: %E\n", FLT_MAX );
   printf("Precision value: %d\n", FLT_DIG );

   return 0;
}
```

当您在 Linux 上编译并执行上面的程序时，它会产生下列结果：

```
Storage size for float : 4
Minimum float positive value: 1.175494E-38
Maximum float positive value: 3.402823E+38
Precision value: 6
```

void 类型

void 类型指定没有可用的值。它通常用于以下三种情况下：

类型	描述
函数返回为空	C 中有各种函数都不返回值，或者您可以说它们返回空。不返回值的函数的返回类型为空。例如 **void exit (int status);**
函数参数为空	C 中有各种函数不接受任何参数。不带参数的函数可以接受一个 void。例如 **int rand(void);**
指针指向 **void**	类型为 void 的指针代表对象的地址，而不是类型。例如，内存分配函数 ***void* malloc(size_t size);** 返回指向 void 的指针，可以转换为任何数据类型。

如果现在您还是无法完全理解 void 类型，不用太担心，在后续的章节中我们将会详细讲解这些概念。

C 变量

变量其实只不过是程序可操作的存储区的名称。C 中每个变量都有指定的类型，类型决定了变量存储的大小和布局，值的范围可以存储在内存中，运算符可应用于变量上。

变量的名称可以由字母、数字和下划线字符组成。它必须以字母或下划线开头。大写字母和小写字母是不同的，因为 C 是大小写敏感的。基于前一章讲解的基本类型，将有以下集中基本变量类型：

类型	描述
char	通常是一个八位字节（一个字节）。这是一个整数类型。
int	对机器而言，整数的最自然的大小。
float	单精度浮点值。
double	双精度浮点值。
void	表示类型的缺失。

C 语言也允许定义各种其他类型的变量，比如枚举、指针、数组、结构、共用体等等，这将会在后续的章节中进行讲解，本章节我们先讲解基本变量类型。

C 中的变量定义

变量定义就是告诉编译器在何处创建变量的存储，以及如何创建变量的存储。变量定义指定一个数据类型，并包含了该类型的一个或多个变量的列表，如下所示：

```
type variable_list;
```

在这里，**type** 必须是一个有效的 C 数据类型，可以是 char、w_char、int、float、double、bool 或任何用户自定义的对象，**variable_list** 可以由一个或多个标识符名称组成，多个标识符之间用逗号分隔。下面列出几个有效的声明：

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

行 **int i, j, k;** 声明并定义了变量 i、j 和 k，这指示编译器创建类型为 int 的名为 i、j、k 的变量。

变量可以在声明的时候被初始化（指定一个初始值）在他们的宣言。初始化器由一个等号，后跟一个常量表达式组成，如下所示：

```
type variable_name = value;
```

下面列举几个实例：

```
extern int d = 3, f = 5;    // d 和 f 的声明
int d = 3, f = 5;           // 定义并初始化 d 和 f
byte z = 22;                // 定义并初始化 z
char x = 'x';               // 变量 x 的值为 'x'
```

不带初始化的定义：带有静态存储持续时间的变量会被隐式初始化为 NULL（所有字节的值都是 0），其他所有变量的初始值是未定义的。

C 中的变量声明

变量声明向编译器保证变量以给定的类型和名称存在，这样编译器在不需要知道变量完整细节的情况下也能继续进一步的编译。变量声明只在编译时有它的意义，在程序连接时编译器需要实际的变量声明。

当您使用多个文件且只在其中一个文件中定义变量时（定义变量的文件在程序连接时是可用的），变量声明就显得非常有用。您可以使用 **extern** 关键字在任何地方声明一个变量。虽然您可以在程序中多次声明一个变量，但变量只能在某个文件、函数或代码块中被定义一次。

实例

尝试下面的实例，其中，变量在头部就已经被声明，但它们是在主函数内被定义和初始化的：

```
#include <stdio.h>

// 变量声明
extern int a, b;
extern int c;
extern float f;

int main ()
{
  /* 变量定义 */
  int a, b;
  int c;
  float f;

  /* 实际初始化 */
  a = 10;
  b = 20;

  c = a + b;
  printf("value of c : %d \n", c);

  f = 70.0/3.0;
  printf("value of f : %f \n", f);

  return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of c : 30
value of f : 23.333334
```

同样的，在函数声明时，提供一个函数名，而函数的实际定义则可以在任何地方进行。例如：

```
// 函数声明
int func();

int main()
{
    // 函数调用
    int i = func();
}

// 函数定义
int func()
{
    return 0;
}
```

C 中的左值（Lvalues）和右值（Rvalues）

C 中有两种类型的表达式：

1. 左值（**lvalue**）：指向内存位置的表达式被称为左值（lvalue）表达式。左值可以出现在赋值号的左边或右边。
2. 右值（**rvalue**）：术语右值（rvalue）指的是存储在内存中某些地址的数值。右值是不能

对其进行赋值的表达式，也就是说，右值可以出现在赋值号的右边，但不能出现在赋值号的左边。

变量是左值，因此可以出现在赋值号的左边。数值型的字面值是右值，因此不能被赋值，不能出现在赋值号的左边。下面是一个有效的语句：

```
int g = 20;
```

但是下面这个就不是一个有效的语句，会生成编译时错误：

```
10 = 20;
```

C 常量

常量是固定值，在程序执行期间不会改变。这些固定的值，又叫做字面量。

常量可以是任何的基本数据类型，比如整数常量、浮点常量、字符常量，或字符串字面值，也有枚举常量。

常量就像是常规的变量，只不过常量的值在定义后不能进行修改。

整数常量

整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：0x 或 0X 表示十六进制，0 表示八进制，不带前缀则默认表示十进制。

整数常量也可以带一个后缀，后缀是 U 和 L 的组合，U 表示无符号整数（unsigned），L 表示长整数（long）。后缀可以是大写，也可以是小写，U 和 L 的顺序任意。

下面列举几个整数常量的实例：

```
212         /* 合法的 */
215u        /* 合法的 */
0xFeeL      /* 合法的 */
078         /* 非法的：8 不是八进制的数字 */
032UU       /* 非法的：不能重复后缀 */
```

以下是各种类型的整数常量的实例：

```
85         /* 十进制 */
0213       /* 八进制 */
0x4b       /* 十六进制 */
30         /* 整数 */
30u        /* 无符号整数 */
30l        /* 长整数 */
30ul       /* 无符号长整数 */
```

浮点常量

浮点常量由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。

当使用小数形式表示时，必须包含小数点、指数，或同时包含两者。当使用指数形式表示时，必须包含整数部分、小数部分，或同时包含两者。带符号的指数是用 e 或 E 引入的。

下面列举几个浮点常量的实例：

```
3.14159       /* 合法的 */
314159E-5L    /* 合法的 */
510E          /* 非法的：不完整的指数 */
210f          /* 非法的：没有小数或指数 */
.e55          /* 非法的：缺少整数或分数 */
```

字符常量

字符常量是括在单引号中，例如，'x' 可以存储在 **char** 类型的简单变量中。

字符常量可以是一个普通的字符（例如 'x'）、一个转义序列（例如 '\t'），或一个通用的字符（例如 '\u02C0'）。

在 C 中，有一些特定的字符，当它们前面有反斜杠时，它们就具有特殊的含义，被用来表示如换行符（\n）或制表符（\t）等。下表列出了一些这样的转义序列码：

转义序列	含义
\	\ 字符
\'	' 字符
\"	" 字符
\?	? 字符
\a	警报铃声
\b	退格键
\f	换页符
\n	换行符
\r	回车
\t	水平制表符
\v	垂直制表符
\ooo	一到三位的八进制数
\xhh . . .	一个或多个数字的十六进制数

下面的实例显示了一些转义序列字符：

```
#include <stdio.h>

int main()
{
   printf("Hello\tWorld\n\n");

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Hello   World
```

字符串常量

字符串字面值或常量是括在双引号 "" 中的。一个字符串包含类似于字符常量的字符：普通的字符、转义序列和通用的字符。

您可以使用空格做分隔符，把一个很长的字符串常量进行分行。

下面的实例显示了一些字符串常量。下面这三种形式所显示的字符串是相同的。

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

定义常量

在 C 中，有两种简单的定义常量的方式：

1. 使用 **#define** 预处理器。
2. 使用 **const** 关键字。

#define 预处理器

下面是使用 #define 预处理器定义常量的形式：

```
#define identifier value
```

具体请看下面的实例：

```
#include <stdio.h>

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main()
{

   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of area : 50
```

const 关键字

您可以使用 **const** 前缀声明指定类型的常量，如下所示：

```
const type variable = value;
```

具体请看下面的实例：

```
#include <stdio.h>

int main()
{
   const int  LENGTH = 10;
   const int  WIDTH  = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of area : 50
```

请注意，把常量定义为大写字母形式，是一个很好的编程实践。

C 存储类

存储类定义 C 程序中变量/函数的范围（可见性）和生命周期。这些说明符放置在它们所修饰的类型之前。下面列出 C 程序中可用的存储类：

- auto
- register
- static
- extern

auto 存储类

auto 存储类是所有局部变量默认的存储类。

```
{
   int mount;
   auto int month;
}
```

上面的实例定义了两个带有相同存储类的变量，auto 只能用在函数内，即 auto 只能修饰局部变量。

register 存储类

register 存储类用于定义存储在寄存器中而不是 RAM 中的局部变量。这意味着变量的最大尺寸等于寄存器的大小（通常是一个词），且不能对它应用一元的 '&' 运算符（因为它没有内存位置）。

```
{
   register int  miles;
}
```

寄存器只用于需要快速访问的变量，比如计数器。还应注意的是，定义 'register' 并不意味着变量将被存储在寄存器中，它意味着变量可能存储在寄存器中，这取决于硬件和实现的限制。

static 存储类

static 存储类指示编译器在程序的生命周期内保持局部变量的存在，而不需要在每次它进入和离开作用域时进行创建和销毁。因此，使用 static 修饰局部变量可以在函数调用之间保持局部变量的值。

static 修饰符也可以应用于全局变量。当 static 修饰全局变量时，会使变量的作用域限制在声明它的文件内。

在 C 编程中，当 **static** 用在类数据成员上时，会导致仅有一个该成员的副本被类的所有对象共享。

```
#include <stdio.h>

/* 函数声明 */
void func(void);

static int count = 5; /* 全局变量 */

main()
{
   while(count--)
   {
      func();
   }
   return 0;
}
/* 函数定义 */
void func( void )
{
   static int i = 5; /* 局部静态变量 */
   i++;

   printf("i is %d and count is %d\n", i, count);
}
```

可能您现在还无法理解这个实例，因为我已经使用了函数和全局变量，这两个概念目前为止还没进行讲解。即使您现在不能完全理解，也没有关系，后续的章节我们会详细讲解。当上面的代码被编译和执行时，它会产生下列结果：

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

extern 存储类

extern 存储类用于提供一个全局变量的引用，它对所有的程序文件都是可见的。当您使用 'extern' 时，对于无法初始化的变量，会把变量名指向一个之前定义过的存储位置。

当您有多个文件且定义了一个可以在其他文件中使用的全局变量或函数时，可以在其他文件中使用 *extern* 来得到已定义的变量或函数的引用。可以这么理解，*extern* 是用来在另一个文件中声明一个全局变量或函数。

extern 修饰符通常用于当有两个或多个文件共享相同的全局变量或函数的时候，如下所示：

第一个文件：**main.c**

```
#include <stdio.h>

int count ;
extern void write_extern();

main()
{
   count = 5;
   write_extern();
}
```

第二个文件：**support.c**

```
#include <stdio.h>

extern int count;

void write_extern(void)
{
   printf("count is %d\n", count);
}
```

在这里，第二个文件中的 *extern* 关键字用于声明已经在第一个文件 main.c 中定义的 *count*。现在，编译这两个文件，如下所示：

```
 $gcc main.c support.c
```

这会产生 **a.out** 可执行程序，当程序被执行时，它会产生下列结果：

```
5
```

C 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C 语言内置了丰富的运算符，并提供了以下类型的运算符：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 杂项运算符

本教程将逐一介绍算术运算符、关系运算符、逻辑运算符、按位运算符、赋值运算符和其他运算符。

算术运算符

下表显示了 C 语言支持的所有算术运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
--	自减运算符，整数值减少 1	A-- 将得到 9

实例

请看下面的实例，了解 C 语言中所有可用的算术运算符：

```
#include <stdio.h>

main()
{
   int a = 21;
   int b = 10;
   int c ;

   c = a + b;
   printf("Line 1 - c 的值是 %d\n", c );
   c = a - b;
   printf("Line 2 - c 的值是 %d\n", c );
   c = a * b;
   printf("Line 3 - c 的值是 %d\n", c );
   c = a / b;
   printf("Line 4 - c 的值是 %d\n", c );
   c = a % b;
   printf("Line 5 - c 的值是 %d\n", c );
   c = a++;
   printf("Line 6 - c 的值是 %d\n", c );
   c = a--;
   printf("Line 7 - c 的值是 %d\n", c );

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 31
Line 2 - c 的值是 11
Line 3 - c 的值是 210
Line 4 - c 的值是 2
Line 5 - c 的值是 1
Line 6 - c 的值是 21
Line 7 - c 的值是 22
```

关系运算符

下表显示了 C 语言支持的所有关系运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(A == B) 不为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(A != B) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(A > B) 不为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(A < B) 为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	(A >= B) 不为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	(A <= B) 为真。

实例

请看下面的实例，了解 C 语言中所有可用的关系运算符：

```
#include <stdio.h>

main()
{
   int a = 21;
   int b = 10;
   int c ;

   if( a == b )
   {
      printf("Line 1 - a 等于 b\n" );
   }
   else
   {
      printf("Line 1 - a 不等于 b\n" );
   }
   if ( a < b )
   {
      printf("Line 2 - a 小于 b\n" );
   }
   else
   {
      printf("Line 2 - a 不小于 b\n" );
   }
   if ( a > b )
   {
      printf("Line 3 - a 大于 b\n" );
   }
   else
   {
      printf("Line 3 - a 不大于 b\n" );
   }
   /* 改变 a 和 b 的值 */
   a = 5;
   b = 20;
   if ( a <= b )
   {
      printf("Line 4 - a 小于或等于 b\n" );
   }
   if ( b >= a )
   {
      printf("Line 5 - b 大于或等于 b\n" );
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - a 不等于 b
Line 2 - a 不小于 b
Line 3 - a 大于 b
Line 4 - a 小于或等于 b
Line 5 - b 大于或等于 b
```

逻辑运算符

下表显示了 C 语言支持的所有关系逻辑运算符。假设变量 **A** 的值为 1，变量 **B** 的值为 0，则：

运算符	描述	实例
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(A && B) 为假。
||	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A || B) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!(A && B) 为真。

实例

请看下面的实例，了解 C 语言中所有可用的逻辑运算符：

```
#include <stdio.h>

main()
{
   int a = 5;
   int b = 20;
   int c ;

   if ( a && b )
   {
      printf("Line 1 - 条件为真\n" );
   }
   if ( a || b )
   {
      printf("Line 2 - 条件为真\n" );
   }
   /* 改变 a 和 b 的值 */
   a = 0;
   b = 10;
   if ( a && b )
   {
      printf("Line 3 - 条件为真\n" );
   }
   else
   {
      printf("Line 3 - 条件不为真\n" );
   }
   if ( !(a && b) )
   {
      printf("Line 4 - 条件为真\n" );
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - 条件为真
Line 2 - 条件为真
Line 3 - 条件不为真
Line 4 - 条件为真
```

位运算符

位运算符作用于位，并逐位执行操作。&、 | 和 ^ 的真值表如下所示：

p	q	p & q	p	q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

假设如果 A = 60，且 B = 13，现在以二进制格式表示，它们如下所示：

```
A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011
```

下表显示了 C 语言支持的位运算符。假设变量 **A** 的值为 60，变量 **B** 的值为 13，则：

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	(A & B) 将得到 12，即为 0000 1100
		如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(A | B) 将得到 61，即为 0011 1101
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	(A ^ B) 将得到 49，即为 0011 0001
~	二进制补码运算符是一元运算符，具有"翻转"位效果。	(~A) 将得到 -61，即为 1100 0011，2 的补码形式，带符号的二进制数。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	A << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	A >> 2 将得到 15，即为 0000 1111

实例

请看下面的实例，了解 C 语言中所有可用的位运算符：

```
#include <stdio.h>

main()
{

   unsigned int a = 60;    /* 60 = 0011 1100 */
   unsigned int b = 13;    /* 13 = 0000 1101 */
   int c = 0;

   c = a & b;       /* 12 = 0000 1100 */
   printf("Line 1 - c 的值是 %d\n", c );

   c = a | b;       /* 61 = 0011 1101 */
   printf("Line 2 - c 的值是 %d\n", c );

   c = a ^ b;       /* 49 = 0011 0001 */
   printf("Line 3 - c 的值是 %d\n", c );

   c = ~a;          /*-61 = 1100 0011 */
   printf("Line 4 - c 的值是 %d\n", c );

   c = a << 2;     /* 240 = 1111 0000 */
   printf("Line 5 - c 的值是 %d\n", c );

   c = a >> 2;     /* 15 = 0000 1111 */
   printf("Line 6 - c 的值是 %d\n", c );
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 12
Line 2 - c 的值是 61
Line 3 - c 的值是 49
Line 4 - c 的值是 -61
Line 5 - c 的值是 240
Line 6 - c 的值是 15
```

赋值运算符

下表列出了 C 语言支持的赋值运算符：

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给左边操作数	C = A + B 将把 A + B 的值赋给 C
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	C += A 相当于 C = C + A
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	C -= A 相当于 C = C - A
*=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	C = A 相当于 C = C A
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	C /= A 相当于 C = C / A
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	C %= A 相当于 C = C % A
<<=	左移且赋值运算符	C <<= 2 等同于 C = C << 2
>>=	右移且赋值运算符	C >>= 2 等同于 C = C >> 2
&=	按位与且赋值运算符	C &= 2 等同于 C = C & 2
^=	按位异或且赋值运算符	C ^= 2 等同于 C = C ^ 2
|=	按位或且赋值运算符	C |= 2 等同于 C = C | 2

实例

请看下面的实例，了解 C 语言中所有可用的赋值运算符：

```
#include <stdio.h>

main()
{
   int a = 21;
   int c ;

   c =  a;
   printf("Line 1 - =  运算符实例，c 的值 = %d\n", c );

   c +=  a;
   printf("Line 2 - += 运算符实例，c 的值 = %d\n", c );

   c -=  a;
   printf("Line 3 - -= 运算符实例，c 的值 = %d\n", c );

   c *=  a;
   printf("Line 4 - *= 运算符实例，c 的值 = %d\n", c );

   c /=  a;
   printf("Line 5 - /= 运算符实例，c 的值 = %d\n", c );

   c  = 200;
   c %=  a;
   printf("Line 6 - %= 运算符实例，c 的值 = %d\n", c );

   c <<=  2;
   printf("Line 7 - <<= 运算符实例，c 的值 = %d\n", c );

   c >>=  2;
   printf("Line 8 - >>= 运算符实例，c 的值 = %d\n", c );

   c &=  2;
   printf("Line 9 - &= 运算符实例，c 的值 = %d\n", c );

   c ^=  2;
   printf("Line 10 - ^= 运算符实例，c 的值 = %d\n", c );

   c |=  2;
   printf("Line 11 - |= 运算符实例，c 的值 = %d\n", c );

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - =  运算符实例，c 的值 = 21
Line 2 - += 运算符实例，c 的值 = 42
Line 3 - -= 运算符实例，c 的值 = 21
Line 4 - *= 运算符实例，c 的值 = 441
Line 5 - /= 运算符实例，c 的值 = 21
Line 6 - %= 运算符实例，c 的值 = 11
Line 7 - <<= 运算符实例，c 的值 = 44
Line 8 - >>= 运算符实例，c 的值 = 11
Line 9 - &= 运算符实例，c 的值 = 2
Line 10 - ^= 运算符实例，c 的值 = 0
Line 11 - |= 运算符实例，c 的值 = 2
```

杂项运算符 ? sizeof & 三元

下表列出了 C 语言支持的其他一些重要的运算符，包括 **sizeof** 和 **? :**。

运算符	描述	实例
sizeof()	返回变量的大小。	sizeof(a) 将返回 4，其中 a 是整数。
&	返回变量的地址。	&a; 将给出变量的实际地址。
*	指向一个变量。	*a; 将指向一个变量。
? :	条件表达式	如果条件为真 ? 则值为 X : 否则值为 Y

实例

请看下面的实例，了解 C 语言中所有可用的杂项运算符：

```
#include <stdio.h>

main()
{
   int a = 4;
   short b;
   double c;
   int* ptr;

   /* sizeof 运算符实例 */
   printf("Line 1 - 变量 a 的大小 = %d\n", sizeof(a) );
   printf("Line 2 - 变量 b 的大小 = %d\n", sizeof(b) );
   printf("Line 3 - 变量 c 的大小 = %d\n", sizeof(c) );

   /* & 和 * 运算符实例 */
   ptr = &a;    /* 'ptr' 现在包含 'a' 的地址 */
   printf("a 的值是 %d\n", a);
   printf("*ptr 是 %d\n", *ptr);

   /* 三元运算符实例 */
   a = 10;
   b = (a == 1) ? 20: 30;
   printf( "b 的值是 %d\n", b );

   b = (a == 10) ? 20: 30;
   printf( "b 的值是 %d\n", b );
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值是 4
*ptr 是 4
b 的值是 30
b 的值是 20
```

C 中的运算符优先级

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 x = 7 + 3 2，在这里，*x* 被赋值为 *13*，而不是 *20*，因为运算符 具有比 + 更高的优先级，所以首先计算乘法 3*2，然后再加上 7。

下表将按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格的上面，具有较低优先级的运算符出现在表格的下面。在表达式中，较高优先级的运算符会优先被计算。

类别	运算符	结合性
后缀	() [] -> . ++ - -	从左到右
一元	+ - ! ~ ++ - - (type)* & sizeof	从右到左
乘除	* / %	从左到右
加减	+ -	从左到右
移位	<< >>	从左到右
关系	< <= > >=	从左到右
相等	== !=	从左到右
位与 AND	&	从左到右
位异或 XOR	^	从左到右
位或 OR	|	从左到右
逻辑与 AND	&&	从左到右
逻辑或 OR	||	从左到右
条件	?:	从右到左
赋值	= += -= *= /= %=>>= <<= &= ^= |=	从右到左
逗号	,	从左到右

实例

请看下面的实例，了解 C 语言中运算符的优先级：

```
#include <stdio.h>

main()
{
   int a = 20;
   int b = 10;
   int c = 15;
   int d = 5;
   int e;

   e = (a + b) * c / d;      // ( 30 * 15 ) / 5
   printf("(a + b) * c / d 的值是 %d\n",  e );

   e = ((a + b) * c) / d;    // (30 * 15 ) / 5
   printf("((a + b) * c) / d 的值是 %d\n" ,  e );

   e = (a + b) * (c / d);   // (30) * (15/5)
   printf("(a + b) * (c / d) 的值是 %d\n",  e );

   e = a + (b * c) / d;     //  20 + (150/5)
   printf("a + (b * c) / d 的值是 %d\n" ,  e );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

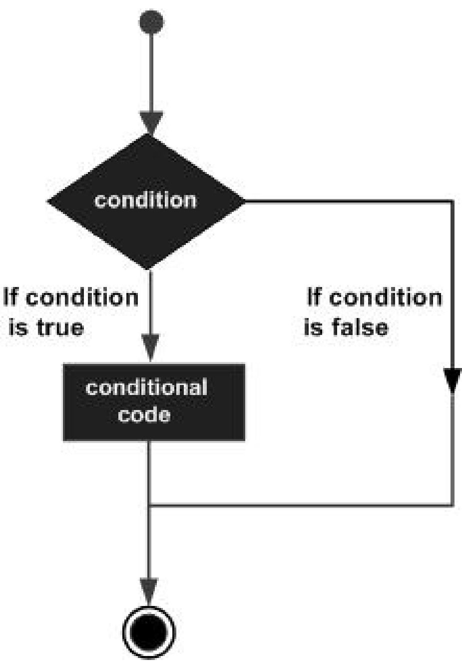
```
(a + b) * c / d 的值是 90
((a + b) * c) / d 的值是 90
(a + b) * (c / d) 的值是 90
a + (b * c) / d 的值是 50
```

C 判断

判断结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。

C 语言把任何非零和非空的值假定为 **true**，把零或 **null** 假定为 **false**。

下面是大多数编程语言中典型的判断结构的一般形式：



判断语句

C 语言提供了以下类型的判断语句。点击链接查看每个语句的细节。

语句	描述
if 语句	一个 **if 语句** 由一个布尔表达式后跟一个或多个语句组成。
if...else 语句	一个 **if 语句** 后可跟一个可选的 **else 语句**，else 语句在布尔表达式为假时执行。
嵌套 if 语句	您可以在一个 **if** 或 **else if** 语句内使用另一个 **if** 或 **else if** 语句。
switch 语句	一个 **switch** 语句允许测试一个变量等于多个值时的情况。
嵌套 switch 语句	您可以在一个 **switch** 语句内使用另一个 **switch** 语句。

? : 运算符

我们已经在前面的章节中讲解了 条件运算符 **? :**，可以用来替代 **if...else** 语句。它的一般形式如下：

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。

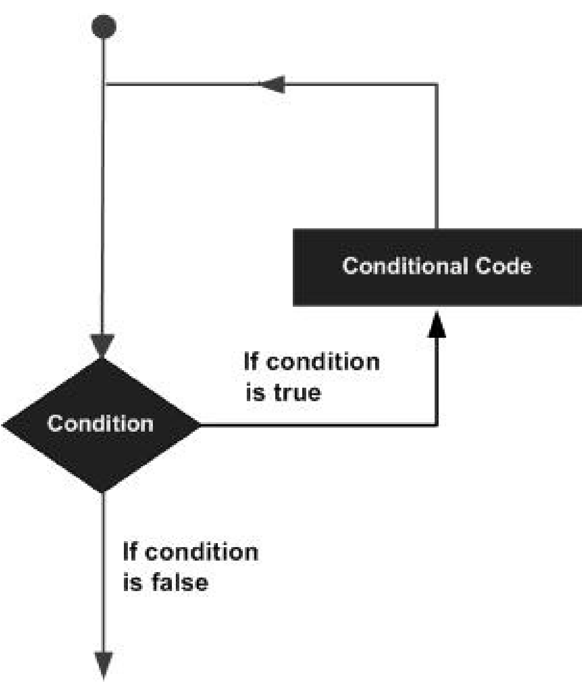
? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个 ? 表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个 ? 表达式的值。

C 循环

有的时候，可能需要多次执行同一块代码。一般情况下，语句是顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了允许更为复杂的执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的一般形式：



循环类型

C 语言提供了以下几种循环类型。点击链接查看每个类型的细节。

循环类型	描述
while 循环	当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。
for 循环	多次执行一个语句序列，简化管理循环变量的代码。
do...while 循环	除了它是在循环主体结尾测试条件外，其他与 while 语句类似。
嵌套循环	您可以在 while、for 或 do..while 循环内使用一个或多个循环。

循环控制语句

循环控制语句更改执行的正常序列。当执行离开一个范围时，所有在该范围中创建的自动对象都会被销毁。

C 提供了下列的控制语句。点击链接查看每个语句的细节。

控制语句	描述
break 语句	终止 **loop** 或 **switch** 语句，程序流将继续执行紧接着 loop 或 switch 的下一条语句。
continue 语句	引起循环跳过主体的剩余部分，立即重新开始测试条件。
goto 语句	将控制转移到被标记的语句。但是不建议在程序中使用 goto 语句。

无限循环

如果条件永远不为假，则循环将变成无限循环。**for** 循环在传统意义上可用于实现无限循环。由于构成循环的三个表达式中任何一个都不是必需的，您可以将某些条件表达式留空来构成一个无限循环。

```
#include <stdio.h>

int main ()
{

   for( ; ; )
   {
      printf("This loop will run forever.\n");
   }

   return 0;
}
```

当条件表达式不存在时，它被假设为真。您也可以设置一个初始值和增量表达式，但是一般情况下，C 程序员偏向于使用 for(;;) 结构来表示一个无限循环。

注意：您可以按 Ctrl + C 键终止一个无限循环。

C 函数

函数是一组一起执行任务的语句。每个 C 程序都至少有一个函数，即主函数 **main()** ，所有简单的程序都可以定义其他额外的函数。

您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。

函数声明告诉编译器函数的名称、返回类型和参数。函数定义提供了函数的实际主体。

C 标准库提供了大量的程序可以调用的内置函数。例如，函数 **strcat()** 用来连接两个字符串，函数 **memcpy()** 用来复制内存到另一个位置。

函数还有很多叫法，比如方法、子例程或程序，等等。

定义函数

C 语言中的函数定义的一般形式如下：

```
return_type function_name( parameter list )
{
   body of the function
}
```

在 C 语言中，函数由一个函数头和一个函数主体组成。下面列出一个函数的所有组成部分：

- 返回类型：一个函数可以返回一个值。**return_type** 是函数返回的值的数据类型。有些函数执行所需的操作而不返回值，在这种情况下，return_type 是关键字 **void**。
- 函数名称：这是函数的实际名称。函数名和参数列表一起构成了函数签名。
- 参数：参数就像是占位符。当函数被调用时，您向参数传递一个值，这个值被称为实际参数。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。
- 函数主体：函数主体包含一组定义函数执行任务的语句。

实例

以下是 **max()** 函数的源代码。该函数有两个参数 num1 和 num2，会返回这两个数中较大的那个数：

```
/* 函数返回两个数中较大的那个数 */
int max(int num1, int num2)
{
   /* 局部变量声明 */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

函数声明

函数声明会告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

函数声明包括以下几个部分：

```
return_type function_name( parameter list );
```

针对上面定义的函数 max()，以下是函数声明：

```
int max(int num1, int num2);
```

在函数声明中，参数的名称并不重要，只有参数的类型是必需的，因此下面也是有效的声明：

```
int max(int, int);
```

当您在一个源文件中定义函数且在另一个文件中调用函数时，函数声明是必需的。在这种情况下，您应该在调用函数的文件顶部声明函数。

调用函数

创建 C 函数时，会定义函数做什么，然后通过调用函数来完成已定义的任务。

当程序调用函数时，程序控制权会转移给被调用的函数。被调用的函数执行已定义的任务，当函数的返回语句被执行时，或到达函数的结束括号时，会把程序控制权交还给主程序。

调用函数时，传递所需参数，如果函数返回一个值，则可以存储返回值。例如：

```
#include <stdio.h>

/* 函数声明 */
int max(int num1, int num2);

int main ()
{
   /* 局部变量定义 */
   int a = 100;
   int b = 200;
   int ret;

   /* 调用函数来获取最大值 */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}

/* 函数返回两个数中较大的那个数 */
int max(int num1, int num2)
{
   /* 局部变量声明 */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

把 max() 函数和 main() 函数放一块，编译源代码。当运行最后的可执行文件时，会产生下列结果：

```
Max value is : 200
```

函数参数

如果函数要使用参数，则必须声明接受参数值的变量。这些变量称为函数的形式参数。

形式参数就像函数内的其他局部变量，在进入函数时被创建，退出函数时被销毁。

当调用函数时，有两种向函数传递参数的方式：

调用类型	描述
传值调用	该方法把参数的实际值复制给函数的形式参数。在这种情况下，修改函数内的形式参数不会影响实际参数。
引用调用	该方法把参数的地址复制给形式参数。在函数内，该地址用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。

默认情况下，C 使用传值调用来传递参数。一般来说，这意味着函数内的代码不能改变用于调用函数的实际参数。

C 作用域规则

任何一种编程中，作用域是程序中定义的变量所存在的区域，超过该区域变量就不能被访问。C 语言中有三个地方可以声明变量：

1. 在函数或块内部的局部变量
2. 在所有函数外部的全局变量
3. 在形式参数的函数参数定义中

让我们来看看什么是局部变量、全局变量和形式参数。

局部变量

在某个函数或块的内部声明的变量称为局部变量。它们只能被该函数或该代码块内部的语句使用。局部变量在函数外部是不可知的。下面是使用局部变量的实例。在这里，所有的变量 a、b 和 c 是 main() 函数的局部变量。

```
#include <stdio.h>

int main ()
{
  /* 局部变量声明 */
  int a, b;
  int c;

  /* 实际初始化 */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

全局变量

全局变量是定义在函数外部，通常是在程序的顶部。全局变量在整个程序生命周期内都是有效的，在任意的函数内部能访问全局变量。

全局变量可以被任何函数访问。也就是说，全局变量在声明后整个程序中都是可用的。下面是使用全局变量和局部变量的实例：

```
#include <stdio.h>

/* 全局变量声明 */
int g;

int main ()
{
  /* 局部变量声明 */
  int a, b;

  /* 实际初始化 */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

在程序中，局部变量和全局变量的名称可以相同，但是在函数内，局部变量的值会覆盖全局变量的值。下面是一个实例：

```
#include <stdio.h>

/* 全局变量声明 */
int g = 20;

int main ()
{
  /* 局部变量声明 */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of g = 10
```

形式参数

函数的参数，形式参数，被当作该函数内的局部变量，它们会优先覆盖全局变量。下面是一个实例：

```
#include <stdio.h>

/* 全局变量声明 */
int a = 20;

int main ()
{
  /* 在主函数中的局部变量声明 */
  int a = 10;
  int b = 20;
  int c = 0;

  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);

  return 0;
}

/* 添加两个整数的函数 */
int sum(int a, int b)
{
    printf ("value of a in sum() = %d\n",  a);
    printf ("value of b in sum() = %d\n",  b);

    return a + b;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

初始化局部变量和全局变量

当局部变量被定义时，系统不会对其初始化，您必须自行对其初始化。定义全局变量时，系统会自动对其初始化，如下所示：

数据类型	初始化默认值
int	0
char	'\0'
float	0
double	0
pointer	NULL

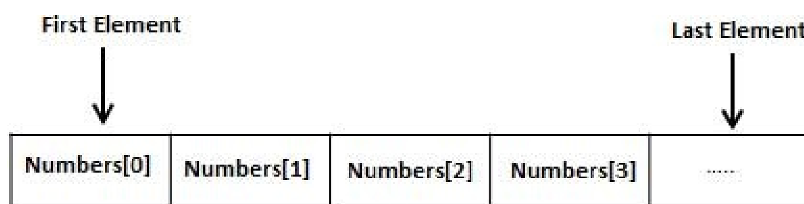
正确地初始化变量是一个良好的编程习惯，否则程序可能会产生意外的结果，因为未初始化的变量会导致一些在内存位置中已经可用的垃圾值。

C 数组

C 语言支持数组数据结构，它可以存储一个固定大小的相同类型元素的顺序集合。数组是用来存储一系列数据，但它往往被认为是一系列相同类型的变量。

数组的声明并不是声明一个个单独的变量，比如 number0、number1、...、number99，而是声明一个数组变量，比如 numbers，然后使用 numbers[0]、numbers[1]、...、numbers[99] 来代表一个个单独的变量。数组中的特定元素可以通过索引访问。

所有的数组都是由连续的内存位置组成。最低的地址对应第一个元素，最高的地址对应最后一个元素。



声明数组

在 C 中要声明一个数组，需要指定元素的类型和元素的数量，如下所示：

```
type arrayName [ arraySize ];
```

这叫做一维数组。**arraySize** 必须是一个大于零的整数常量，**type** 可以是任意有效的 C 数据类型。例如，要声明一个类型为 double 的包含 10 个元素的数组 **balance**，声明语句如下：

```
double balance[10];
```

现在 *balance* 是一个可用的数组，可以容纳 10 个类型为 double 的数字。

初始化数组

在 C 中，您可以逐个初始化数组，也可以使用一个初始化语句，如下所示：

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

大括号 { } 之间的值的数目不能大于我们在数组声明时在方括号 [ ] 中指定的元素数目。

如果您省略掉了数组的大小，数组的大小则为初始化时元素的个数。因此，如果：

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

您将创建一个数组，它与前一个实例中所创建的数组是完全相同的。下面是一个为数组中某个元素赋值的实例：

```
balance[4] = 50.0;
```

上述的语句把数组中第五个元素的值赋为 50.0。所有的数组都是以 0 作为它们第一个元素的索引，也被称为基索引，数组的最后一个索引是数组的总大小减去 1。以下是上面所讨论的数组的的图形表示：

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

访问数组元素

数组元素可以通过数组名称加索引进行访问。元素的索引是放在方括号内，跟在数组名称的后边。例如：

```
double salary = balance[9];
```

上面的语句将把数组中第 10 个元素的值赋给 salary 变量。下面的实例使用了上述的三个概念，即，声明数组、数组赋值、访问数组：

```
#include <stdio.h>

int main ()
{
   int n[ 10 ]; /* n 是一个包含 10 个整数的数组 */
   int i,j;

   /* 初始化数组元素 */
   for ( i = 0; i < 10; i++ )
   {
      n[ i ] = i + 100; /* 设置元素 i 为 i + 100 */
   }

   /* 输出数组中每个元素的值 */
   for (j = 0; j < 10; j++ )
   {
      printf("Element[%d] = %d\n", j, n[j] );
   }

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

C 中数组详解

在 C 中，数组是非常重要的，且需要了解更多的细节。下面列出了 C 程序员必须清楚的一些与数组相关的重要概念：

概念	描述
多维数组	C 支持多维数组。多维数组最简单的形式是二维数组。
传递数组给函数	您可以通过指定不带索引的数组名称来给函数传递一个指向数组的指针。
从函数返回数组	C 允许从函数返回数组。
指向数组的指针	您可以通过指定不带索引的数组名称来生成一个指向数组中第一个元素的指针。

C 指针

学习 C 语言的指针既简单又有趣。通过指针，可以简化一些 C 编程任务的执行，还有一些任务，如动态内存分配，没有指针是无法执行的。所以，想要成为一名优秀的 C 程序员，学习指针是很有必要的。

正如您所知道的，每一个变量都有一个内存位置，每一个内存位置都定义了可使用连字号（&）运算符访问的地址，它表示了在内存中的一个地址。考虑下面的实例，它将输出定义的变量地址：

```
#include <stdio.h>

int main ()
{
   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

通过上面的实例，我们了解了什么是内存地址以及如何访问它。接下来让我们看看什么是指针。

什么是指针？

指针是一个变量，其值为另一个变量的地址，即，内存位置的直接地址。就像其他变量或常量一样，您必须在使用指针存储其他变量地址之前，对其进行声明。指针变量声明的一般形式为：

```
type *var-name;
```

在这里，**type** 是指针的基类型，它必须是一个有效的 C 数据类型，**var-name** 是指针变量的名称。用来声明指针的星号 * 与乘法中使用的星号是相同的。但是，在这个语句中，星号是用来指定一个变量是指针。以下是有效的指针声明：

```
int    *ip;    /* 一个整型的指针 */
double *dp;    /* 一个 double 型的指针 */
float  *fp;    /* 一个浮点型的指针 */
char   *ch     /* 一个字符型的指针 */
```

所有指针的值的实际数据类型，不管是整行、浮点型、字符型，还是其他的数据类型，都是一样的，都是一个代表内存地址的长的十六进制数。不同数据类型的指针之间唯一的不同是，指针所指向的变量或常量的数据类型。

如何使用指针？

使用指针时会频繁进行以下几个操作：定义一个指针变量、把变量地址赋值给指针、访问指针变量中可用地址的值。这些是通过使用一元运算符 * 来返回位于操作数所指定地址的变量的值。下面的实例使用了这些操作：

```
#include <stdio.h>

int main ()
{
   int  var = 20;   /* 实际变量的声明 */
   int  *ip;        /* 指针变量的声明 */

   ip = &var;  /* 在指针变量中存储 var 的地址 */

   printf("Address of var variable: %x\n", &var  );

   /* 在指针变量中存储的地址 */
   printf("Address stored in ip variable: %x\n", ip );

   /* 使用指针访问值 */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

C 中的 NULL 指针

在变量声明的时候，为指针变量赋一个 NULL 值是一个良好的编程习惯，以防没有确切的地址可以赋值的情况。赋为 NULL 值的指针被称为空指针。

NULL 指针是一个定义在标准库中的值为零的常量。请看下面的程序：

```
#include <stdio.h>

int main ()
{
   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
The value of ptr is 0
```

在大多数的操作系统上，程序不允许访问地址为 0 的内存，因为该内存是操作系统保留的。然而，内存地址 0 有特别重要的意义，这表明 NULL 指针不指向一个可访问的内存位置。但按照惯例，如果指针包含空值（零值），则假定它不指向任何东西。

如需检查一个空指针，您可以使用 if 语句，如下所示：

```
if(ptr)     /* 如果 p 非空，则完成 */
if(!ptr)    /* 如果 p 为空，则完成 */
```

C 指针详解

在 C 中，有很多指针相关的概念，这些概念都很简单，但是都很重要。下面列出了 C 程序员必须清楚的一些与指针相关的重要概念：

概念	描述
指针的算术运算	可以对指针进行四种算术运算：++、--、+、-
指针数组	可以定义用来存储指针的数组。
指向指针的指针	C 允许指向指针的指针。
传递指针给函数	通过引用或地址传递参数能使传递的参数在调用函数中被改变。
从函数返回指针	C 允许函数返回指针到局部变量、静态变量和动态内存分配。

C 字符串

在 C 语言中，字符串实际上是使用 **null** 字符 '\0' 终止的一维字符数组。因此，一个以 null 结尾的字符串，包含了组成字符串的字符。

下面的声明和初始化创建了一个 "Hello" 字符串。由于在数组的末尾存储了空字符，所以字符数组的大小比单词 "Hello" 的字符数多一个。

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

依据数组初始化规则，您可以把上面的语句写成以下语句：

```
char greeting[] = "Hello";
```

以下是 C/C++ 中定义的字符串的内存表示：

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

其实，您不需要把 *null* 字符放在字符串常量的末尾。C 编译器会在初始化数组时，自动把 '\0' 放在字符串的末尾。让我们尝试输出上面的字符串：

```
#include <stdio.h>

int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

   printf("Greeting message: %s\n", greeting );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Greeting message: Hello
```

C 中有大量操作字符串的函数：

函数	目的
strcpy(s1, s2);	复制字符串 s2 到字符串 s1。
strcat(s1, s2);	链接字符串 s2 到字符串 s1 的末尾。
strlen(s1);	返回字符串 s1 的长度。
strcmp(s1, s2);	如果,s1 和 s2 是相同的，则返回 0；如果 s1s2 则返回大于 0。
strchr(s1, ch);	返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。
strstr(s1, s2);	返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。

下面的实例使用了上述的一些函数：

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[12] = "Hello";
   char str2[12] = "World";
   char str3[12];
   int  len ;

   /* 复制 str1 到 str3 */
   strcpy(str3, str1);
   printf("strcpy( str3, str1) :  %s\n", str3 );

   /* 连接 str1 和 str2 */
   strcat( str1, str2);
   printf("strcat( str1, str2):   %s\n", str1 );

   /* 连接后，str1 的总长度 */
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

您可以在 C 标准库中找到更多字符串相关的函数。

C 结构体

C 数组允许定义可存储相同类型数据项的变量的类型，结构是 C 编程中另一种用户自定义的可用的数据类型，它允许您存储不同类型的数据项。

结构用于表示一条记录，假设您想要跟踪图书馆中书本的动态，您可能需要跟踪每本书的下列属性：

- Title
- Author
- Subject
- Book ID

定义结构

为了定义结构，您必须使用 **struct** 语句。struct 语句定义了一个包含多个成员的新的数据类型，struct 语句的格式如下：

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

structure tag 是可选的，每个 member definition 是标准的变量定义，比如 int i; 或者 float f; 或者其他有效的变量定义。在结构定义的末尾，最后一个分号之前，您可以指定一个或多个结构变量，这是可选的。下面是声明 Book 结构的方式：

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

访问结构成员

为了访问结构的成员，我们使用成员访问运算符（.）。成员访问运算符是结构变量名称和我们要访问的结构成员之间的一个句号。您可以使用 **struct** 关键字来定义结构类型的变量。下面的实例演示了结构的用法：

```
#include <stdio.h>
#include <string.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   struct Books Book1;        /* 声明 Book1，类型为 Book */
   struct Books Book2;        /* 声明 Book2，类型为 Book */

   /* Book1 详述 */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* Book2 详述 */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* 输出 Book1 信息 */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* 输出 Book2 信息 */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

结构作为函数参数

您可以把结构作为函数参数，传参方式与其他类型的变量或指针类似。您可以使用上面实例中的方式来访问结构变量：

```
#include <stdio.h>
#include <string.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* 函数声明 */
void printBook( struct Books book );
int main( )
{
   struct Books Book1;        /* 声明 Book1，类型为 Book */
   struct Books Book2;        /* 声明 Book2，类型为 Book */

   /* Book1 详述 */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* Book2 详述 */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* 输出 Book1 信息 */
   printBook( Book1 );

   /* 输出 Book2 信息 */
   printBook( Book2 );

   return 0;
}
void printBook( struct Books book )
{
   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

指向结构的指针

您可以定义指向结构的指针，方式与定义指向其他类型变量的指针相似，如下所示：

```
struct Books *struct_pointer;
```

现在，您可以在上述定义的指针变量中存储结构变量的地址。为了查找结构变量的地址，请把 & 运算符放在结构名称的前面，如下所示：

```
struct_pointer = &Book1;
```

为了使用指向该结构的指针访问结构的成员，您必须使用 -> 运算符，如下所示：

```
struct_pointer->title;
```

让我们使用结构指针来重写上面的实例，这将有助于您理解结构指针的概念：

```
#include <stdio.h>
#include <string.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* 函数声明 */
void printBook( struct Books *book );
int main( )
{
   struct Books Book1;        /* 声明 Book1，类型为 Book */
   struct Books Book2;        /* 声明 Book2，类型为 Book */

   /* Book1 详述 */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* Book2 详述 */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* 通过传 Book1 的地址来输出 Book1 信息 */
   printBook( &Book1 );

   /* 通过传 Book2 的地址来输出 Book2 信息 */
   printBook( &Book2 );

   return 0;
}
void printBook( struct Books *book )
{
   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

位域

有些信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用 1 位二进位即可。为了节省存储空间，并使处理简便，C 语言又提供了一种数据结构，称为"位域"或"位段"。

所谓"位域"是把一个字节中的二进位划分为几个不同的区域，并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。

典型的实例：

- 用 1 位二进位存放一个开关量时，只有 0 和 1 两种状态。
- 读取外部文件格式——可以读取非标准的文件格式。例如：9 位的整数。

位域的定义和位域变量的说明

位域定义与结构定义相仿，其形式为：

```
    struct 位域结构名
        { 位域列表 };
```

其中位域列表的形式为：

```
    类型说明符 位域名: 位域长度
```

例如：

```
struct bs{
    int a:8;
    int b:2;
    int c:6;
};
```

位域变量的说明与结构变量说明的方式相同。 可采用先定义后说明，同时定义说明或者直接说明这三种方式。例如：

```
struct bs{
    int a:8;
    int b:2;
    int c:6;
}data;
```

说明 data 为 bs 变量，共占两个字节。其中位域 a 占 8 位，位域 b 占 2 位，位域 c 占 6 位。

让我们再来看一个实例：

```
struct packed_struct {
  unsigned int f1:1;
  unsigned int f2:1;
  unsigned int f3:1;
  unsigned int f4:1;
  unsigned int type:4;
  unsigned int my_int:9;
} pack;
```

在这里，packed_struct 包含了 6 个成员：四个 1 位的标识符 f1..f4、一个 4 位的 type 和一个 9 位的 my_int。

对于位域的定义尚有以下几点说明：

- 一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。例如：

  ```
  struct bs{
      unsigned a:4;
      unsigned  :4;    /* 空域 */
      unsigned b:4;    /* 从下一单元开始存放 */
      unsigned c:4
  }
  ```

  在这个位域定义中，a 占第一字节的 4 位，后 4 位填 0 表示不使用，b 从第二字节开始，占用 4 位，c 占用 4 位。

- 由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说不能超过8位二进位。如果最大长度大于计算机的整数字长，一些编译器可能会允许域的内存重叠，另外一些编译器可能会把大于一个域的部分存储在下一个字中。

- 位域可以是无名位域，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：

  ```
  struct k{
      int a:1;
      int  :2;    /* 该 2 位不能使用 */
      int b:3;
      int c:2;
  };
  ```

从以上分析可以看出，位域在本质上就是一种结构类型，不过其成员是按二进位分配的。

位域的使用

位域的使用和结构成员的使用相同，其一般形式为：

```
位域变量名·位域名
```

位域允许用各种格式输出。

请看下面的实例：

```
main(){
    struct bs{
        unsigned a:1;
        unsigned b:3;
        unsigned c:4;
    } bit,*pbit;
    bit.a=1;    /* 给位域赋值（应注意赋值不能超过该位域的允许范围） */
    bit.b=7;    /* 给位域赋值（应注意赋值不能超过该位域的允许范围） */
    bit.c=15;    /* 给位域赋值（应注意赋值不能超过该位域的允许范围） */
    printf("%d,%d,%d\n",bit.a,bit.b,bit.c);    /* 以整型量格式输出三个域的内容 */
    pbit=&bit;    /* 把位域变量 bit 的地址送给指针变量 pbit */
    pbit->a=0;    /* 用指针方式给位域 a 重新赋值，赋为 0 */
    pbit->b&=3;    /* 使用了复合的位运算符 "&="，相当于：pbit->b=pbit->b&3，位域 b 中原有值为 7，
    pbit->c|=1;    /* 使用了复合位运算符"|="，相当于：pbit->c=pbit->c|1，其结果为 15 */
    printf("%d,%d,%d\n",pbit->a,pbit->b,pbit->c);    /* 用指针方式输出了这三个域的值 */
}
```

上例程序中定义了位域结构 bs，三个位域为 a、b、c。说明了 bs 类型的变量 bit 和指向 bs 类型的指针变量 pbit。这表示位域也是可以使用指针的。

C 共用体

共用体是一种特殊的数据类型，允许您在相同的内存位置存储不同的数据类型。您可以定义一个带有多成员的共用体，但是任何时候只能有一个成员带有值。共用体提供了一种使用相同的内存位置的有效方式。

定义共用体

为了定义结构体，您必须使用 **union** 语句，方式与定义结构类似。union 语句定义了一个新的数据类型，带有多个成员。union 语句的格式如下：

```
union [union tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

union tag 是可选的，每个 member definition 是标准的变量定义，比如 int i; 或者 float f; 或者其他有效的变量定义。在共用体定义的末尾，最后一个分号之前，您可以指定一个或多个共用体变量，这是可选的。下面定义一个名为 Data 的共用体类型，有三个成员 i、f 和 str：

```
union Data
{
   int i;
   float f;
   char  str[20];
} data;
```

现在，**Data** 类型的变量可以存储一个整数、一个浮点数，或者一个字符串。这意味着一个变量（相同的内存位置）可以存储多个多种类型的数据。您可以根据需要在一个共用体内使用任何内置的或者用户自定义的数据类型。

共用体占用的内存应足够存储共用体中最大的成员。例如，在上面的实例中，Data 将占用 20 个字节的内存空间，因为在各个成员中，字符串所占用的空间是最大的。下面的实例将显示上面的共用体占用的总内存大小：

```
#include <stdio.h>
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Memory size occupied by data : 20
```

访问共用体成员

为了访问共用体的成员，我们使用成员访问运算符（.）。成员访问运算符是共用体变量名称和我们要访问的共用体成员之间的一个句号。您可以使用 **union** 关键字来定义共用体类型的变量。下面的实例演示了共用体的用法：

```
#include <stdio.h>
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

在这里，我们可以看到共用体的 **i** 和 **f** 成员的值有损坏，因为最后赋给变量的值占用了内存位置，这也是 **str** 成员能够完好输出的原因。现在让我们再来看一个相同的实例，这次我们在同一时间只使用一个变量，这也演示了使用共用体的主要目的：

```
#include <stdio.h>
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

在这里，所有的成员都能完好输出，因为同一时间只用到一个成员。

C 位域

如果程序的结构中包含多个开关量，只有 TRUE/FALSE 变量，如下：

```
struct
{
  unsigned int widthValidated;
  unsigned int heightValidated;
} status;
```

这种结构需要 8 字节的内存空间，但在实际上，在每个变量中，我们只存储 0 或 1。在这种情况下，C 语言提供了一中更好的利用内存空间的方式。如果您在结构内使用这样的变量，您可以定义变量的宽度来告诉编译器，您将只使用这些字节。例如，上面的结构可以重写成：

```
struct
{
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status;
```

现在，上面的结构中，status 变量将占用 4 个字节的内存空间，但是只有 2 位被用来存储值。如果您用了 32 个变量，每一个变量宽度为 1 位，那么 status 结构将使用 4 个字节，但只要您再多用一个变量，如果使用了 33 个变量，那么它将分配内存的下一段来存储第 33 个变量，这个时候就开始使用 8 个字节。让我们看看下面的实例来理解这个概念：

```
#include <stdio.h>
#include <string.h>

/* 定义简单的结构 */
struct
{
  unsigned int widthValidated;
  unsigned int heightValidated;
} status1;

/* 定义位域结构 */
struct
{
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status2;

int main( )
{
   printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
   printf( "Memory size occupied by status2 : %d\n", sizeof(status2));

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

位域声明

在结构内声明位域的形式如下：

```
struct
{
  type [member_name] : width ;
};
```

下面是有关位域中变量元素的描述：

元素	描述
type	整数类型，决定了如何解释位域的值。类型可以是整型、有符号整型、无符号整型。
member_name	位域的名称。
width	位域中位的数量。宽度必须小于或等于指定类型的位宽度。

带有预定义宽度的变量被称为位域。位域可以存储多于 1 位的数，例如，需要一个变量来存储从 0 到 7 的值，您可以定义一个宽度为 3 位的位域，如下：

```
struct
{
  unsigned int age : 3;
} Age;
```

上面的结构定义指示 C 编译器，age 变量将只使用 3 位来存储这个值，如果您试图使用超过 3 位，则无法完成。让我们来看下面的实例：

```
#include <stdio.h>
#include <string.h>

struct
{
  unsigned int age : 3;
} Age;

int main( )
{
   Age.age = 4;
   printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
   printf( "Age.age : %d\n", Age.age );

   Age.age = 7;
   printf( "Age.age : %d\n", Age.age );

   Age.age = 8;
   printf( "Age.age : %d\n", Age.age );

   return 0;
}
```

当上面的代码被编译时，它会带有警告，当上面的代码被执行时，它会产生下列结果：

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

C typedef

C 语言提供了 **typedef** 关键字，您可以使用它来为类型取一个新的名字。下面的实例为单字节数字定义了一个术语 **BYTE**：

```
typedef unsigned char BYTE;
```

在这个类型定义之后，标识符 BYTE 可作为类型 **unsigned char** 的缩写，例如：

```
BYTE  b1, b2;
```

按照惯例，定义时会大写字母，以便提醒用户类型名称是一个象征性的缩写，但您也可以使用小写字母，如下：

```
typedef unsigned char byte;
```

您也可以使用 **typedef** 来为用户自定义的数据类型取一个新的名字。例如，您可以对结构体使用 typedef 来定义一个新的数据类型，然后使用这个新的数据类型来直接定义结构变量，如下：

```
#include <stdio.h>
#include <string.h>

typedef struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} Book;

int main( )
{
   Book book;

   strcpy( book.title, "C Programming");
   strcpy( book.author, "Nuha Ali");
   strcpy( book.subject, "C Programming Tutorial");
   book.book_id = 6495407;

   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book  title : C Programming
Book  author : Nuha Ali
Book  subject : C Programming Tutorial
Book  book_id : 6495407
```

typedef vs #define

#define 是 C 指令，用于为各种数据类型定义别名，与 **typedef** 类似，但是它们有以下几点不同：

- **typedef** 仅限于为类型定义符号名称，**#define** 不仅可以为类型定义别名，也能为数值定义别名，比如您可以定义 1 为 ONE。
- **typedef** 是由编译器执行解释的，**#define** 语句是由预编译器进行处理的。

下面是 #define 的最简单的用法：

```
#include <stdio.h>

#define TRUE  1
#define FALSE 0

int main( )
{
   printf( "Value of TRUE : %d\n", TRUE);
   printf( "Value of FALSE : %d\n", FALSE);

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of TRUE : 1
Value of FALSE : 0
```

C 输入 & 输出

当我们提到输入时，这意味着要向程序填充一些数据。输入可以是以文件的形式或从命令行中进行。C 语言提供了一系列内置的函数来读取给定的输入，并根据需要填充到程序中。

当我们提到输出时，这意味着要在屏幕上、打印机上或任意文件中显示一些数据。C 语言提供了一系列内置的函数来输出数据到计算机屏幕上和保存数据到文本文件或二进制文件中。

标准文件

C 语言把所有的设备都当作文件。所以设备（比如显示器）被处理的方式与文件相同。以下三个文件会在程序执行时自动打开，以便访问键盘和屏幕。

标准文件	文件指针	设备
标准输入	stdin	键盘
标准输出	stdout	屏幕
标准错误	stderr	您的屏幕

文件指针是访问文件的方式，本节将讲解如何从屏幕读取值以及如何把结果输出到屏幕上。

getchar() & putchar() 函数

int getchar(void) 函数从屏幕读取下一个可用的字符，并把它返回为一个整数。这个函数在同一个时间内只会读取一个单一的字符。您可以在循环内使用这个方法，以便从屏幕上读取多个字符。

int putchar(int c) 函数把字符输出到屏幕上，并返回相同的字符。这个函数在同一个时间内只会输出一个单一的字符。您可以在循环内使用这个方法，以便在屏幕上输出多个字符。

请看下面的实例：

```
#include <stdio.h>
int main( )
{
   int c;

   printf( "Enter a value :");
   c = getchar( );

   printf( "\nYou entered: ");
   putchar( c );

   return 0;
}
```

当上面的代码被编译和执行时，它会等待您输入一些文本，当您输入一个文本并按下回车键时，程序会继续并只会读取一个单一的字符，显示如下：

```
$./a.out
<b>Enter a value :</b> this is test
<b>You entered:</b> t
```

gets() & puts() 函数

char *gets(char* s) 函数从 **stdin** 读取一行到 **s** 所指向的缓冲区，直到一个终止符或 EOF。

int puts(const char *s) 函数把字符串 s 和一个尾随的换行符写入到 **stdout**。

```
#include <stdio.h>
int main( )
{
   char str[100];

   printf( "Enter a value :");
   gets( str );

   printf( "\nYou entered: ");
   puts( str );

   return 0;
}
```

当上面的代码被编译和执行时，它会等待您输入一些文本，当您输入一个文本并按下回车键时，程序会继续并读取一整行直到该行结束，显示如下：

```
$./a.out
<b>Enter a value :</b> this is test
<b>You entered:</b> This is test
```

scanf() 和 printf() 函数

int scanf(const char *format, ...) 函数从标准输入流 **stdin** 读取输入，并根据提供的 **format** 来浏览输入。

int printf(const char *format, ...) 函数把输出写入到标准输出流 **stdout** ，并根据提供的格式产生输出。

format 可以是一个简单的常量字符串，但是您可以分别指定 %s、%d、%c、%f 等来输出或读取字符串、整数、字符或浮点数。还有许多其他可用的格式选项，可以根据需要使用。如需了解完整的细节，可以查看这些函数的参考手册。现在让我们通过下面这个简单的实例来加深理解：

```
#include <stdio.h>
int main( )
{
   char str[100];
   int i;

   printf( "Enter a value :");
   scanf("%s %d", str, &i);

   printf( "\nYou entered: %s %d ", str, i);

   return 0;
}
```

当上面的代码被编译和执行时，它会等待您输入一些文本，当您输入一个文本并按下回车键时，程序会继续并读取输入，显示如下：

```
$./a.out
<b>Enter a value :</b> seven 7
<b>You entered:</b> seven 7
```

在这里，应当指出的是，scanf() 期待输入的格式与您给出的 %s 和 %d 相同，这意味着您必须提供有效的输入，比如 "string integer"，如果您提供的是 "string string" 或 "integer integer"，它会被认为是错误的输入。另外，在读取字符串时，只要遇到一个空格，scanf() 就会停止读取，所以 "this is test" 对 scanf() 来说是三个字符串。

C 文件读写

上一章我们讲解了 C 语言处理的标准输入和输出设备。本章我们将介绍 C 程序员如何创建、打开、关闭文本文件或二进制文件。

一个文件，无论它是文本文件还是二进制文件，都是代表了一系列的字节。C 语言不仅提供了访问顶层的函数，也提供了底层（OS）调用来处理存储设备上的文件。本章将讲解文件管理的重要调用。

打开文件

您可以使用 **fopen( )** 函数来创建一个新的文件或者打开一个已有的文件，这个调用会初始化类型 **FILE** 的一个对象，类型 **FILE** 包含了所有用来控制流的必要的信息。下面是这个函数调用的原型：

```
FILE *fopen( const char * filename, const char * mode );
```

在这里，**filename** 是字符串，用来命名文件，访问模式 **mode** 的值可以是下列值中的一个：

模式	描述
r	打开一个已有的文本文件，允许读取文件。
w	打开一个文本文件，允许写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会从文件的开头写入内容。
a	打开一个文本文件，以追加模式写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会在已有的文件内容中追加内容。
r+	打开一个文本文件，允许读写文件。
w+	打开一个文本文件，允许读写文件。如果文件已存在，则文件会被截断为零长度，如果文件不存在，则会创建一个新文件。
a+	打开一个文本文件，允许读写文件。如果文件不存在，则会创建一个新文件。读取会从文件的开头开始，写入则只能是追加模式。

如果处理的是二进制文件，则需使用下面的访问模式来取代上面的访问模式：

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

关闭文件

;

为了关闭文件，请使用 fclose( ) 函数。函数的原型如下：

```
int fclose( FILE *fp );
```

如果成功关闭文件，**fclose( )** 函数返回零，如果关闭文件时发生错误，函数返回 **EOF**。这个函数实际上，会清空缓冲区中的数据，关闭文件，并释放用于该文件的所有内存。EOF 是一个定义在头文件 **stdio.h** 中的常量。

C 标准库提供了各种函数来按字符或者以固定长度字符串的形式读写文件。

写入文件

下面是把字符写入到流中的最简单的函数：

```
int fputc( int c, FILE *fp );
```

函数 **fputc()** 把参数 c 的字符值写入到 fp 所指向的输出流中。如果写入成功，它会返回写入的字符，如果发生错误，则会返回 **EOF**。您可以使用下面的函数来把一个以 null 结尾的字符串写入到流中：

```
int fputs( const char *s, FILE *fp );
```

函数 **fputs()** 把字符串 **s** 写入到 fp 所指向的输出流中。如果写入成功，它会返回一个非负值，如果发生错误，则会返回 **EOF**。您也可以使用 **int fprintf(FILE *fp,const char* format, ...)** 函数来写把一个字符串写入到文件中。尝试下面的实例：

注意：请确保您有可用的 **/tmp** 目录，如果不存在该目录，则需要在您的计算机上先创建该目录。

```
#include <stdio.h>

main()
{
   FILE *fp;

   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

当上面的代码被编译和执行时，它会在 /tmp 目录中创建一个新的文件 **test.txt**，并使用两个不同的函数写入两行。接下来让我们来读取这个文件。

读取文件

下面是从文件读取单个字符的最简单的函数：

```
int fgetc( FILE * fp );
```

fgetc() 函数从 fp 所指向的输入文件中读取一个字符。返回值是读取的字符，如果发生错误则返回 **EOF**。下面的函数允许您从流中读取一个字符串：

```
char *fgets( char *buf, int n, FILE *fp );
```

函数 **fgets()** 从 fp 所指向的输入流中读取 n - 1 个字符。它会把读取的字符串复制到缓冲区 **buf**，并在最后追加一个 **null** 字符来终止字符串。

如果这个函数在读取最后一个字符之前就遇到一个换行符 '\n' 或文件的末尾 EOF，则只会返回读取到的字符，包括换行符。您也可以使用 **int fscanf(FILE *fp, const char* format, ...)** 函数来从文件中读取字符串，但是在遇到第一个空格字符时，它会停止读取。

```
#include <stdio.h>

main()
{
   FILE *fp;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

当上面的代码被编译和执行时，它会读取上一部分创建的文件，产生下列结果：

```
1 : This
2: is testing for fprintf...

3: This is testing for fputs...
```

首先，**fscanf()** 方法只读取了 **This**，因为它在后边遇到了一个空格。其次，调用 **fgets()** 读取剩余的部分，直到行尾。最后，调用 **fgets()** 完整地读取第二行。

二进制 I/O 函数

下面两个函数用于二进制输入和输出：

```
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
```

这两个函数都是用于存储块的读写 - 通常是数组或结构体。

C 预处理器

C 预处理器不是编译器的组成部分，但是它是编译过程中一个单独的步骤。简言之，C 预处理器只不过是一个文本替换工具而已，它们会指示编译器在实际编译之前完成所需的预处理。我们将把 C 预处理器（C Preprocessor）简写为 CPP。

所有的预处理器命令都是以井号（#）开头。它必须是第一个非空字符，为了增强可读性，预处理器指令应从第一列开始。下面列出了所有重要的预处理器指令：

指令	描述
#define	定义宏
#include	包含一个源代码文件
#undef	取消已定义的宏
#ifdef	如果宏已经定义，则返回真
#ifndef	如果宏没有定义，则返回真
#if	如果给定条件为真，则编译下面代码
#else	#if 的替代方案
#elif	如果前面的 #if 给定条件不为真，当前条件为真，则编译下面代码
#endif	结束一个 #if……#else 条件编译块
#error	当遇到标准错误时，输出错误消息
#pragma	使用标准化方法，向编译器发布特殊的命令到编译器中

预处理器实例

分析下面的实例来理解不同的指令。

```
#define MAX_ARRAY_LENGTH 20
```

这个指令告诉 CPP 把所有的 MAX*ARRAY_LENGTH* 替换为 *20*。使用 #define_ 定义常量来增强可读性。

```
#include <stdio.h>
#include "myheader.h"
```

这些指令告诉 CPP 从系统库中获取 stdio.h，并添加文本到当前的源文件中。下一行告诉 CPP 从本地目录中获取 **myheader.h**，并添加内容到当前的源文件中。

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

这个指令告诉 CPP 取消已定义的 FILE_SIZE，并定义它为 42。

```
#ifndef MESSAGE
   #define MESSAGE "You wish!"
#endif
```

这个指令告诉 CPP 只有当 MESSAGE 未定义时，才定义 MESSAGE。

```
#ifdef DEBUG
   /* Your debugging statements here */
#endif
```

这个指令告诉 CPP 如果定义了 DEBUG，则执行处理语句。在编译时，如果您向 gcc 编译器传递了 *-DDEBUG* 开关量，这个指令就非常有用。它定义了 DEBUG，您可以在编译期间随时开启或关闭调试。

预定义宏

ANSI C 定义了许多宏。在编程中您可以使用这些宏，但是不同直接修改这些预定义的宏。

宏	描述
__DATE__	当前日期，一个以 "MMM DD YYYY" 格式表示的字符常量。
__TIME__	当前时间，一个以 "HH:MM:SS" 格式表示的字符常量。
__FILE__	这会包含当前文件名，一个字符串常量。
__LINE__	这会包含当前行号，一个十进制常量。
__STDC__	当编译器以 ANSI 标准编译时，则定义为 1。

让我们来尝试下面的实例：

```
#include <stdio.h>

main()
{
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );

}
```

当上面的代码（在文件 **test.c** 中）被编译和执行时，它会产生下列结果：

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

预处理器运算符

C 预处理器提供了下列的运算符来帮助您创建宏：

宏延续运算符（\）

一个宏通常写在一个单行上。但是如果宏太长，一个单行容纳不下，则使用宏延续运算符（\）。例如：

```
#define  message_for(a, b)  \
    printf(#a " and " #b ": We love you!\n")
```

字符串常量化运算符（**#**）

在宏定义中，当需要把一个宏的参数转换为字符串常量时，则使用字符串常量化运算符（#）。在宏中使用的该运算符有一个特定的参数或参数列表。例如：

```
#include <stdio.h>

#define  message_for(a, b)  \
    printf(#a " and " #b ": We love you!\n")

int main(void)
{
   message_for(Carole, Debra);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Carole and Debra: We love you!
```

标记粘贴运算符（**##**）

宏定义内的标记粘贴运算符（##）会合并两个参数。它允许在宏定义中两个独立的标记被合并为一个标记。例如：

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void)
{
   int token34 = 40;

   tokenpaster(34);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
token34 = 40
```

这是怎么发生的，因为这个实例会从编译器产生下列的实际输出：

```
printf ("token34 = %d", token34);
```

这个实例演示了 token##n 会连接到 token34 中，在这里，我们使用了字符串常量化运算符（**#**）和标记粘贴运算符（**##**）。

defined() 运算符

预处理器 **defined** 运算符是用在常量表达式中的，用来确定一个标识符是否已经使用 #define 定义过。如果指定的标识符已定义，则值为真（非零）。如果指定的标识符未定义，则值为假（零）。下面的实例演示了 defined() 运算符的用法：

```
#include <stdio.h>

#if !defined (MESSAGE)
   #define MESSAGE "You wish!"
#endif

int main(void)
{
   printf("Here is the message: %s\n", MESSAGE);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Here is the message: You wish!
```

参数化的宏

CPP 一个强大的功能是可以使用参数化的宏来模拟函数。例如，下面的代码是计算一个数的平方：

```
int square(int x) {
   return x * x;
}
```

我们可以使用宏重写上面的代码，如下：

```
#define square(x) ((x) * (x))
```

在使用带有参数的宏之前，必须使用 **#define** 指令定义。参数列表是括在圆括号内，且必须紧跟在宏名称的后边。宏名称和左圆括号之间不允许有空格。例如：

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)
{
   printf("Max between 20 and 10 is %d\n", MAX(10, 20));
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Max between 20 and 10 is 20
```

C 头文件

头文件是扩展名为 **.h** 的文件，包含了 C 函数声明和宏定义，被多个源文件中引用共享。有两种类型的头文件：程序员编写的头文件和编译器自带的头文件。

在程序中要使用头文件，需要使用 C 预处理指令 **#include** 来引用它。前面我们已经看过 **stdio.h** 头文件，它是编译器自带的头文件。

引用头文件相当于复制头文件的内容，但是我们不会直接在源文件中复制头文件的内容，因为这么做很容易出错，特别在程序是由多个源文件组成的时候。

A simple practice in C 或 C++ 程序中，建议把所有的常量、宏、系统全局变量和函数原型写在头文件中，在需要的时候随时引用这些头文件。

引用头文件的语法

使用预处理指令 **#include** 可以引用用户和系统头文件。它的形式有以下两种：

```
#include <file>
```

这种形式用于引用系统头文件。它在系统目录的标准列表中搜索名为 file 的文件。在编译源代码时，您可以通过 -I 选项把目录前置在该列表前。

```
#include "file"
```

这种形式用于引用用户头文件。它在包含当前文件的目录中搜索名为 file 的文件。在编译源代码时，您可以通过 -I 选项把目录前置在该列表前。

引用头文件的操作

#include 指令会指示 C 预处理器浏览指定的文件作为输入。预处理器的输出包含了已经生成的输出，被引用文件生成的输出以及 **#include** 指令之后的文本输出。例如，如果您有一个头文件 header.h，如下：

```
char *test (void);
```

和一个使用了头文件的主程序 *program.c*，如下：

```
int x;
#include "header.h"

int main (void)
{
   puts (test ());
}
```

编译器会看到如下的令牌流：

```
int x;
char *test (void);

int main (void)
{
   puts (test ());
}
```

只引用一次头文件

如果一个头文件被引用两次，编译器会处理两次头文件的内容，这将产生错误。为了防止这种情况，标准的做法是把文件的整个内容放在条件编译语句中，如下：

```
#ifndef HEADER_FILE
#define HEADER_FILE

the entire header file file

#endif
```

这种结构就是通常所说的包装器 **#ifndef**。当再次引用头文件时，条件为假，因为 HEADER_FILE 已定义。此时，预处理器会跳过文件的整个内容，编译器会忽略它。

有条件引用

有时需要从多个不同的头文件中选择一个引用到程序中。例如，需要指定在不同的操作系统上使用的配置参数。您可以通过一系列条件来实现这点，如下：

```
#if SYSTEM_1
   # include "system_1.h"
#elif SYSTEM_2
   # include "system_2.h"
#elif SYSTEM_3
   ...
#endif
```

但是如果头文件比较多的时候，这么做是很不妥当的，预处理器使用宏来定义头文件的名称。这就是所谓的有条件引用。它不是用头文件的名称作为 **#include** 的直接参数，您只需要使用宏名称代替即可：

```
#define SYSTEM_H "system_1.h"
...
#include SYSTEM_H
```

SYSTEM_H 会扩展，预处理器会查找 system_1.h，就像 **#include** 最初编写的那样。SYSTEM_H 可通过 -D 选项被您的 Makefile 定义。

C 强制类型转换

强制类型转换是把变量从一种类型转换为另一种数据类型。例如，如果您想存储一个 long 类型的值到一个简单的整型中，您需要把 long 类型强制转换为 int 类型。您可以使用**强制类型转换运算符**来把值显式地从一种类型转换为另一种类型，如下所示：

```
(type_name) expression
```

请看下面的实例，使用强制类型转换运算符把一个整数变量除以另一个整数变量，得到一个浮点数：

```
#include <stdio.h>

main()
{
   int sum = 17, count = 5;
   double mean;

   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean );

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of mean : 3.400000
```

这里要注意的是强制类型转换运算符的优先级大于除法，因此 **sum** 的值首先被转换为 **double** 型，然后除以 count，得到一个类型为 double 的值。

类型转换可以是隐式的，由编译器自动执行，也可以是显式的，通过使用**强制类型转换运算符**来指定。在编程时，有需要类型转换的时候都用上强制类型转换运算符，是一种良好的编程习惯。

整数提升

整数提升是指把小于 **int** 或 **unsigned int** 的整数类型转换为 **int** 或 **unsigned int** 的过程。请看下面的实例，在 int 中添加一个字符：

```
#include <stdio.h>

main()
{
   int  i = 17;
   char c = 'c'; /* ascii 值是 99 */
   int sum;

   sum = i + c;
   printf("Value of sum : %d\n", sum );

}
```

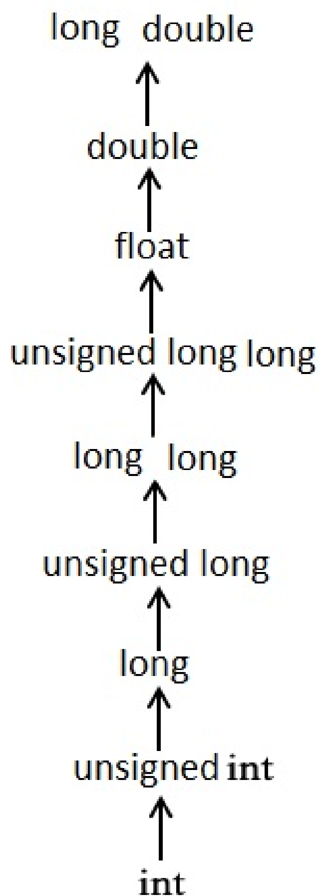
当上面的代码被编译和执行时，它会产生下列结果：

```
Value of sum : 116
```

在这里，sum 的值为 116，因为编译器进行了整数提升，在执行实际加法运算时，把 'c' 的值转换为对应的 ascii 值。

常用的算术转换

常用的算术转换是隐式地把值强制转换为相同的类型。编译器首先执行整数提升，如果操作数类型不同，则它们会被转换为下列层次中出现的最高层次的类型：



常用的算术转换不适用于赋值运算符、逻辑运算符 && 和 ||。让我们看看下面的实例来理解这个概念：

```
#include <stdio.h>

main()
{
   int  i = 17;
   char c = 'c'; /* ascii 值是 99 */
   float sum;

   sum = i + c;
   printf("Value of sum : %f\n", sum );

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of sum : 116.000000
```

在这里，c 首先被转换为整数，但是由于最后的值是 double 型的，所以会应用常用的算术转换，编译器会把 i 和 c 转换为浮点型，并把它们相加得到一个浮点数。

C 错误处理

C 语言不提供对错误处理的直接支持，但是作为一种系统编程语言，它以返回值的形式允许您访问底层数据。在发生错误时，大多数的 C 或 UNIX 函数调用返回 1 或 NULL，同时会设置一个错误代码 **errno**，该错误代码是全局变量，表示在函数调用期间发生了错误。您可以在 <error.h> 头文件中找到各种各样的错误代码。

所以，C 程序员可以通过检查返回值，然后根据返回值决定采取哪种适当的动作。开发人员应该在程序初始化时，把 errno 设置为 0，这是一种良好的编程习惯。0 值表示程序中没有错误。

errno、perror() 和 strerror()

C 语言提供了 **perror()** 和 **strerror()** 函数来显示与 **errno** 相关的文本消息。

- **perror()** 函数显示您传给它的字符串，后跟一个冒号、一个空格和当前 errno 值的文本表示形式。
- **strerror()** 函数，返回一个指针，指针指向当前 errno 值的文本表示形式。

让我们来模拟一种错误情况，尝试打开一个不存在的文件。您可以使用多种方式来输出错误消息，在这里我们使用函数来演示用法。另外有一点需要注意，您应该使用 **stderr** 文件流来输出所有的错误。

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno ;

int main ()
{
   FILE * pf;
   int errnum;
   pf = fopen ("unexist.txt", "rb");
   if (pf == NULL)
   {
      errnum = errno;
      fprintf(stderr, "Value of errno: %d\n", errno);
      perror("Error printed by perror");
      fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
   }
   else
   {
      fclose (pf);
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

被零除的错误

在进行除法运算时，不检查除数是否为零，这是程序员编程时常见的问题，会导致一个运行时错误。

为了避免这种情况发生，下面的代码在进行处罚运算前会先检查除数是否为零：

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   int dividend = 20;
   int divisor = 0;
   int quotient;

   if( divisor == 0){
      fprintf(stderr, "Division by zero! Exiting...\n");
      exit(-1);
   }
   quotient = dividend / divisor;
   fprintf(stderr, "Value of quotient : %d\n", quotient );

   exit(0);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Division by zero! Exiting...
```

程序退出状态

通常情况下，程序成功执行完一个操作正常退出的时候会带有值 EXIT_SUCCESS。在这里，EXIT_SUCCESS 是宏，它被定义为 0。

如果程序中存在一种错误情况，当您退出程序时，会带有状态值 EXIT_FAILURE，被定义为 -1。所以，上面的程序可以写成：

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   int dividend = 20;
   int divisor = 5;
   int quotient;

   if( divisor == 0){
      fprintf(stderr, "Division by zero! Exiting...\n");
      exit(EXIT_FAILURE);
   }
   quotient = dividend / divisor;
   fprintf(stderr, "Value of quotient : %d\n", quotient );

   exit(EXIT_SUCCESS);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of quotient : 4
```

C 递归

递归是以自相似的方式重复项目的处理过程。同样地，在编程语言中，在函数内部调用函数自身，称为递归调用。如下：

```
void recursion()
{
   recursion(); /* 函数调用自身 */
}

int main()
{
   recursion();
}
```

C 语言支持递归，即，一个函数可以调用自身。但在使用递归时，程序员需要注意定义一个从函数退出的条件，否则会进入无限循环。

递归函数在解决许多数学问题上起了至关重要的作用，比如计算一个数的阶乘、生成斐波那契数列，等等。

数的阶乘

下面的实例使用递归函数计算一个给定的数的阶乘：

```
#include <stdio.h>

int factorial(unsigned int i)
{
   if(i <= 1)
   {
      return 1;
   }
   return i * factorial(i - 1);
}
int  main()
{
    int i = 15;
    printf("Factorial of %d is %d\n", i, factorial(i));
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Factorial of 15 is 2004310016
```

斐波那契数列

下面的实例使用递归函数生成一个给定的数的斐波那契数列：

```
#include <stdio.h>

int fibonaci(int i)
{
   if(i == 0)
   {
      return 0;
   }
   if(i == 1)
   {
      return 1;
   }
   return fibonaci(i-1) + fibonaci(i-2);
}

int  main()
{
    int i;
    for (i = 0; i < 10; i++)
    {
       printf("%d\t%n", fibonaci(i));
    }
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
0    1    1    2    3    5    8    13    21    34
```

C 可变参数

有时，您可能会碰到这样的情况，您希望函数带有可变数量的参数，而不是预定义数量的参数。C 语言为这种情况提供了一个解决方案，它允许您定义一个函数，能根据具体的需求接受可变数量的参数。下面的实例演示了这种函数的定义。

```
int func(int, ... )
{
   .
   .
   .
}

int main()
{
   func(1, 2, 3);
   func(1, 2, 3, 4);
}
```

请注意，函数 **func()** 最后一个参数写成省略号，即三个点号（**...**），省略号之前的那个参数总是 **int**，代表了要传递的可变参数的总数。为了使用这个功能，您需要使用 **stdarg.h** 头文件，该文件提供了实现可变参数功能的函数和宏。具体步骤如下：

- 定义一个函数，最后一个参数为省略号，省略号前面的那个参数总是 **int**，表示了参数的个数。
- 在函数定义中创建一个 **va_list** 类型变量，该类型是在 stdarg.h 头文件中定义的。
- 使用 **int** 参数和 **va_start** 宏来初始化 **va_list** 变量为一个参数列表。宏 va_start 是在 stdarg.h 头文件中定义的。
- 使用 **va_arg** 宏和 **va_list** 变量来访问参数列表中的每个项。
- 使用宏 **va_end** 来清理赋予 **va_list** 变量的内存。

现在让我们按照上面的步骤，来编写一个带有可变数量参数的函数，并返回它们的平均值：

```
#include <stdio.h>
#include <stdarg.h>

double average(int num,...)
{

    va_list valist;
    double sum = 0.0;
    int i;

    /* 为 num 个参数初始化 valist */
    va_start(valist, num);

    /* 访问所有赋给 valist 的参数 */
    for (i = 0; i < num; i++)
    {
       sum += va_arg(valist, int);
    }
    /* 清理为 valist 保留的内存 */
    va_end(valist);

    return sum/num;
}

int main()
{
   printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));
   printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}
```

当上面的代码被编译和执行时，它会产生下列结果。应该指出的是，函数 **average()** 被调用两次，每次第一个参数都是表示被传的可变参数的总数。省略号被用来传递可变数量的参数。

```
Average of 2, 3, 4, 5 = 3.500000
Average of 5, 10, 15 = 10.000000
```

C 内存管理

本章将讲解 C 中的动态内存管理。C 语言为内存的分配和管理提供了几个函数。这些函数可以在 **<stdlib.h>** 头文件中找到。

函数	描述
void *calloc(int num, int size);	该函数分配一个带有 function allocates an array of **num** 个元素的数组，每个元素的大小为 **size** 字节。
void free(void *address);	该函数释放 address 所指向的h内存块。
void *malloc(int num);	该函数分配一个 **num** 字节的数组，并把它们进行初始化。
void *realloc(void *address, int newsize);	该函数重新分配内存，把内存扩展到 **newsize**。

动态分配内存

编程时，如果您预先知道数组的大小，那么定义数组时就比较容易。例如，一个存储人名的数组，它最多容纳 100 个字符，所以您可以定义数组，如下所示：

```
char name[100];
```

但是，如果您预先不知道需要存储的文本长度，例如您向存储有关一个主题的详细描述。在这里，我们需要定义一个指针，该指针指向未定义所学内存大小的字符，后续再根据需求来分配内存，如下所示：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   char name[100];
   char *description;

   strcpy(name, "Zara Ali");

   /* 动态分配内存 */
   description = malloc( 200 * sizeof(char) );
   if( description == NULL )
   {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else
   {
      strcpy( description, "Zara ali a DPS student in class 10th");
   }
   printf("Name = %s\n", name );
   printf("Description: %s\n", description );
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```

上面的程序也可以使用 **calloc()** 来编写，只需要把 malloc 替换为 calloc 即可，如下所示：

```
calloc(200, sizeof(char));
```

当动态分配内存时，您有完全控制权，可以传递任何大小的值。而那些预先定义了大小的数组，一旦定义则无法改变大小。

重新调整内存的大小和释放内存

当程序退出时，操作系统会自动释放所有分配给程序的内存，但是，建议您在不需要内存时，都应该调用函数 **free()** 来释放内存。

或者，您可以通过调用函数 **realloc()** 来增加或减少已分配的内存块的大小。让我们使用 realloc() 和 free() 函数，再次查看上面的实例：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   char name[100];
   char *description;

   strcpy(name, "Zara Ali");

   /* 动态分配内存 */
   description = malloc( 30 * sizeof(char) );
   if( description == NULL )
   {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else
   {
      strcpy( description, "Zara ali a DPS student.");
   }
   /* 假设您想要存储更大的描述信息 */
   description = realloc( description, 100 * sizeof(char) );
   if( description == NULL )
   {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else
   {
      strcat( description, "She is in class 10th");
   }

   printf("Name = %s\n", name );
   printf("Description: %s\n", description );

   /* 使用 free() 函数释放内存 */
   free(description);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th
```

您可以尝试一下不重新分配额外的内存，strcat() 函数会生成一个错误，因为存储 description 时可用的内存不足。

C 命令行参数

执行程序时，可以从命令行传值给 C 程序。这些值被称为命令行参数，它们对程序很重要，特别是当您想从外部控制程序，而不是在代码内对这些值进行硬编码时，就显得尤为重要了。

命令行参数是使用 main() 函数参数来处理的，其中，**argc** 是指传入参数的个数，**argv[]** 是一个指针数组，指向传递给程序的每个参数。下面是一个简单的实例，检查命令行是否有提供参数，并根据参数执行相应的动作：

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
   if( argc == 2 )
   {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 )
   {
      printf("Too many arguments supplied.\n");
   }
   else
   {
      printf("One argument expected.\n");
   }
}
```

使用一个参数，编译并执行上面的代码，它会产生下列结果：

```
$./a.out testing
The argument supplied is testing
```

使用两个参数，编译并执行上面的代码，它会产生下列结果：

```
$./a.out testing1 testing2
Too many arguments supplied.
```

不传任何参数，编译并执行上面的代码，它会产生下列结果：

```
$./a.out
One argument expected
```

应当指出的是，**argv[0]** 存储程序的名称，**argv[1]** 是一个指向第一个命令行参数的指针，*argv[n]* 是最后一个参数。如果没有提供任何参数，*argc* 将为 *1*，否则，如果传递了一个参数，**argc* 将被设置为 2。

多个命令行参数之间用空格分隔，但是如果参数本身带有空格，那么传递参数的时候应把参数放置在双引号 "" 或单引号 '' 内部。让我们重新编写上面的实例，有一个空间，那么你可以通过这样的观点，把它们放在双引号或单引号""""。让我们重新编写上面的实例，向程序传递一个放置在双引号内部的命令行参数：

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
   printf("Program name %s\n", argv[0]);

   if( argc == 2 )
   {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 )
   {
      printf("Too many arguments supplied.\n");
   }
   else
   {
      printf("One argument expected.\n");
   }
}
```

使用一个用空格分隔的简单参数，参数括在双引号中，编译并执行上面的代码，它会产生下列结果：

```
$./a.out "testing1 testing2"

Progranm name ./a.out
The argument supplied is testing1 testing2
```

C语言参考

C 标准库 - <assert.h>

简介

C 标准库的 **assert.h**头文件提供了一个名为 **assert** 的宏，它可用于验证程序做出的假设，并在假设为假时输出诊断消息。

已定义的宏 **assert** 指向另一个宏 **NDEBUG**，宏 **NDEBUG** 不是 <assert.h> 的一部分。如果已在引用 <assert.h> 的源文件中定义 NDEBUG 为宏名称，则 **assert** 宏的定义如下：

```
#define assert(ignore) ((void)0)
```

库宏

下面列出了头文件 assert.h 中定义的唯一的函数：

函数	描述
void assert(int expression)	这实际上是一个宏，不是一个函数，可用于在 C 程序中添加诊断。

C 库宏 - assert()

描述

C 库宏 **void assert(int expression)** 允许诊断信息被写入到标准错误文件中。换句话说，它可用于在 C 程序中添加诊断。

声明

下面是 assert() 宏的声明。

```
void assert(int expression);
```

参数

- **expression** -- 这可以是一个变量或任何 C 表达式。如果 **expression** 为 TRUE，assert() 不执行任何动作。如果 **expression** 为 FALSE，assert() 会在标准错误 stderr 上显示错误消息，并中止程序执行。

返回值

这个宏不返回任何值。

实例

下面的实例演示了 assert() 宏的用法。

```
#include <assert.h>
#include <stdio.h>

int main()
{
   int a;
   char str[50];

   printf("请输入一个整数值： ");
   scanf("%d\n", &a);
   assert(a >= 10);
   printf("输入的整数是： %d\n", a);

   printf("请输入字符串： ");
   scanf("%s\n", &str);
   assert(str != NULL);
   printf("输入的字符串是： %s\n", str);

   return(0);
}
```

让我们在交互模式下编译并运行上面的程序，如下所示：

```
请输入一个整数值： 11
输入的整数是： 11
请输入字符串： w3cschool
输入的字符串是： w3cschool
```

C 标准库 - <ctype.h>

简介

C 标准库的 **ctype.h** 头文件提供了一些函数，可用于测试和映射字符。

这些函数接受 **int** 作为参数，它的值必须是 EOF 或表示为一个无符号字符。

如果参数 c 满足描述的条件，则这些函数返回非零（true）。如果参数 c 不满足描述的条件，则这些函数返回零。

库函数

下面列出了头文件 ctype.h 中定义的函数：

函数	描述
int isalnum(int c)	该函数检查所传的字符是否是字母和数字。
int isalpha(int c)	该函数检查所传的字符是否是字母。
int iscntrl(int c)	该函数检查所传的字符是否是控制字符。
int isdigit(int c)	该函数检查所传的字符是否是十进制数字。
int isgraph(int c)	该函数检查所传的字符是否有图形表示法。
int islower(int c)	该函数检查所传的字符是否是小写字母。
int isprint(int c)	该函数检查所传的字符是否是可打印的。
int ispunct(int c)	该函数检查所传的字符是否是标点符号字符。
int isspace(int c)	该函数检查所传的字符是否是空白字符。
int isupper(int c)	该函数检查所传的字符是否是大写字母。
int isxdigit(int c)	该函数检查所传的字符是否是十六进制数字。

标准库还包含了两个转换函数，它们接受并返回一个 "int"

函数	描述
int tolower(int c)	该函数把大写字母转换为小写字母。
int toupper(int c)	该函数把小写字母转换为大写字母。

字符类

字符类	描述
数字	完整的数字集合 { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
十六进制数字	集合 { 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f }
小写字母	集合 { a b c d e f g h i j k l m n o p q r s t u v w x y z }
大写字母	集合 {A B C D E F G H I J K L M N O P Q R S T U V W X Y Z }
字母	小写字母和大写字母的集合
字母数字字符	数字、小写字母和大写字母的集合
标点符号字符	集合 ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~
图形字符	字母数字字符和标点符号字符的集合
空格字符	制表符、换行符、垂直制表符、换页符、回车符、空格符的集合。
可打印字符	字母数字字符、标点符号字符和空格字符的集合。
控制字符	在 ASCII 编码中，这些字符的八进制代码是从 000 到 037，以及 177（DEL）。
空白字符	包括空格符和制表符。
字母字符	小写字母和大写字母的集合。

C 库函数 - isalnum()

描述

C 库函数 **void isalnum(int c)** 检查所传的字符是否是字母和数字。

声明

下面是 isalnum() 函数的声明。

```
int isalnum(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个数字或一个字母，则该函数返回非零值，否则返回 0。

实例

下面的实例演示了 isalnum() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 'd';
   int var2 = '2';
   int var3 = '\t';
   int var4 = ' ';

   if( isalnum(var1) )
   {
      printf("var1 = |%c| 是字母数字\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 不是字母数字\n", var1 );
   }
   if( isalnum(var2) )
   {
      printf("var2 = |%c| 是字母数字\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 不是字母数字\n", var2 );
   }
   if( isalnum(var3) )
   {
      printf("var3 = |%c| 是字母数字\n", var3 );
   }
   else
   {
      printf("var3 = |%c| 不是字母数字\n", var3 );
   }
   if( isalnum(var4) )
   {
      printf("var4 = |%c| 是字母数字\n", var4 );
   }
   else
   {
      printf("var4 = |%c| 不是字母数字\n", var4 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |d| 是字母数字
var2 = |2| 是字母数字
var3 = |	| 不是字母数字
var4 = | | 不是字母数字
```

C 库函数 - isalpha()

描述

C 库函数 **void isalpha(int c)** 检查所传的字符是否是字母。

声明

下面是 isalpha() 函数的声明。

```
int isalpha(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个字母，则该函数返回非零值，否则返回 0。

实例

下面的实例演示了 isalpha() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 'd';
   int var2 = '2';
   int var3 = '\t';
   int var4 = ' ';

   if( isalpha(var1) )
   {
      printf("var1 = |%c| 是一个字母\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 不是一个字母\n", var1 );
   }
   if( isalpha(var2) )
   {
      printf("var2 = |%c| 是一个字母\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 不是一个字母\n", var2 );
   }
   if( isalpha(var3) )
   {
      printf("var3 = |%c| 是一个字母\n", var3 );
   }
   else
   {
      printf("var3 = |%c| 不是一个字母\n", var3 );
   }
   if( isalpha(var4) )
   {
      printf("var4 = |%c| 是一个字母\n", var4 );
   }
   else
   {
      printf("var4 = |%c| 不是一个字母\n", var4 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |d| 是一个字母
var2 = |2| 不是一个字母
var3 = | | 不是一个字母
var4 = | | 不是一个字母
```

C 库函数 - iscntrl()

描述

C 库函数 **void iscntrl(int c)** 检查所传的字符是否是控制字符。

根据标准 ASCII 字符集，控制字符的 ASCII 编码介于 0x00 (NUL) 和 0x1f (US) 之间，以及 0x7f (DEL)，某些平台的特定编译器实现还可以在扩展字符集（0x7f 以上）中定义额外的控制字符。

声明

下面是 iscntrl() 函数的声明。

```
int iscntrl(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个控制字符，则该函数返回非零值，否则返回 0。

实例

下面的实例演示了 iscntrl() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main ()
{
   int i = 0, j = 0;
   char str1[] = "all \a about \t programming";
   char str2[] = "w3cschool \n tutorials";

   /* 输出字符串直到控制字符 \a */
   while( !iscntrl(str1[i]) )
   {
      putchar(str1[i]);
      i++;
   }

   /* 输出字符串直到控制字符 \n */
   while( !iscntrl(str2[j]) )
   {
      putchar(str2[j]);
      j++;
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
all w3cschool
```

C 库函数 - isdigit()

描述

C 库函数 **void isdigit(int c)** 检查所传的字符是否是十进制数字字符。

十进制数字是：0 1 2 3 4 5 6 7 8 9

声明

下面是 isdigit() 函数的声明。

```
int isdigit(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个数字，则该函数返回非零值，否则返回 0。

实例

下面的实例演示了 isdigit() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 'h';
   int var2 = '2';

   if( isdigit(var1) )
   {
      printf("var1 = |%c| 是一个数字\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 不是一个数字\n", var1 );
   }
   if( isdigit(var2) )
   {
      printf("var2 = |%c| 是一个数字\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 不是一个数字\n", var2 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |h| 不是一个数字
var2 = |2| 是一个数字
```

C 库函数 - isgraph()

描述

C 库函数 **void isgraph(int c)** 检查所传的字符是否有图形表示法。

带有图形表示法的字符是除了空白字符（比如 ' '）以外的所有可打印的字符。

声明

下面是 isgraph() 函数的声明。

```
int isgraph(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 有图形表示法，则该函数返回非零值，否则返回 0。

实例

下面的实例演示了 isgraph() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = '3';
   int var2 = 'm';
   int var3 = ' ';

   if( isgraph(var1) )
   {
       printf("var1 = |%c| 是可打印的\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 是不可打印的\n", var1 );
   }
   if( isgraph(var2) )
   {
       printf("var2 = |%c| 是可打印的\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 是不可打印的\n", var2 );
   }
   if( isgraph(var3) )
   {
       printf("var3 = |%c| 是可打印的\n", var3 );
   }
   else
   {
      printf("var3 = |%c| 是不可打印的\n", var3 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |3| 是可打印的
var2 = |m| 是可打印的
var3 = | | 是不可打印的
```

C 库函数 - islower()

描述

C 库函数 **int islower(int c)** 检查所传的字符是否是小写字母。

声明

下面是 islower() 函数的声明。

```
int islower(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个小写字母，则该函数返回非零值（true），否则返回 0（false）。

实例

下面的实例演示了 islower() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 'Q';
   int var2 = 'q';
   int var3 = '3';

   if( islower(var1) )
   {
       printf("var1 = |%c| 是小写字母\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 不是小写字母\n", var1 );
   }
   if( islower(var2) )
   {
       printf("var2 = |%c| 是小写字母\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 不是小写字母\n", var2 );
   }
   if( islower(var3) )
   {
       printf("var3 = |%c| 是小写字母\n", var3 );
   }
   else
   {
      printf("var3 = |%c| 不是小写字母\n", var3 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |Q| 不是小写字母
var2 = |q| 是小写字母
var3 = |3| 不是小写字母
```

C 库函数 - isprint()

描述

C 库函数 **int isprint(int c)** 检查所传的字符是否是可打印的。可打印字符是非控制字符的字符。

声明

下面是 isprint() 函数的声明。

```
int isprint(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个可打印的字符，则该函数返回非零值（true），否则返回 0（false）。

实例

下面的实例演示了 isprint() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 'k';
   int var2 = '8';
   int var3 = '\t';
   int var4 = ' ';

   if( isprint(var1) )
   {
      printf("var1 = |%c| 是可打印的\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 是不可打印的\n", var1 );
   }
   if( isprint(var2) )
   {
      printf("var2 = |%c| 是可打印的\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 是不可打印的\n", var2 );
   }
   if( isprint(var3) )
   {
      printf("var3 = |%c| 是可打印的\n", var3 );
   }
   else
   {
      printf("var3 = |%c| 是不可打印的\n", var3 );
   }
   if( isprint(var4) )
   {
      printf("var4 = |%c| 是可打印的\n", var4 );
   }
   else
   {
      printf("var4 = |%c| 是不可打印的\n", var4 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |k| 是可打印的
var2 = |8| 是可打印的
var3 = |	| 是不可打印的
var4 = | | 是可打印的
```

C 库函数 - ispunct()

描述

C 库函数 **int ispunct(int c)** 检查所传的字符是否是标点符号字符。标点符号字符可以是非字母数字（正如 isalnum 中的一样）的任意图形字符（正如 isgraph 中的一样）。

声明

下面是 ispunct() 函数的声明。

```
int ispunct(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个标点符号字符，则该函数返回非零值（true），否则返回 0（false）。

实例

下面的实例演示了 ispunct() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 't';
   int var2 = '1';
   int var3 = '/';
   int var4 = ' ';

   if( ispunct(var1) )
   {
       printf("var1 = |%c| 是标点符号字符\n", var1 );
   }
   else
   {
       printf("var1 = |%c| 不是标点符号字符\n", var1 );
   }
   if( ispunct(var2) )
   {
       printf("var2 = |%c| 是标点符号字符\n", var2 );
   }
   else
   {
       printf("var2 = |%c| 不是标点符号字符\n", var2 );
   }
   if( ispunct(var3) )
   {
       printf("var3 = |%c| 是标点符号字符\n", var3 );
   }
   else
   {
       printf("var3 = |%c| 不是标点符号字符\n", var3 );
   }
   if( ispunct(var4) )
   {
       printf("var4 = |%c| 是标点符号字符\n", var4 );
   }
   else
   {
       printf("var4 = |%c| 不是标点符号字符\n", var4 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |t| 不是标点符号字符
var2 = |1| 不是标点符号字符
var3 = |/| 是标点符号字符
var4 = | | 不是标点符号字符
```

C 库函数 - isspace()

描述

C 库函数 **int isspace(int c)** 检查所传的字符是否是空白字符。

标准的空白字符包括：

```
' '     (0x20)    space (SPC) 空格符
'\t'    (0x09)    horizontal tab (TAB) 水平制表符
'\n'    (0x0a)    newline (LF) 换行符
'\v'    (0x0b)    vertical tab (VT) 垂直制表符
'\f'    (0x0c)    feed (FF) 换页符
'\r'    (0x0d)    carriage return (CR) 回车符
```

声明

下面是 isspace() 函数的声明。

```
int isspace(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个空白字符，则该函数返回非零值（true），否则返回 0（false）。

实例

下面的实例演示了 isspace() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 't';
   int var2 = '1';
   int var3 = ' ';

   if( isspace(var1) )
   {
       printf("var1 = |%c| 是空白字符\n", var1 );
   }
   else
   {
       printf("var1 = |%c| 不是空白字符\n", var1 );
   }
   if( isspace(var2) )
   {
       printf("var2 = |%c| 是空白字符\n", var2 );
   }
   else
   {
       printf("var2 = |%c| 不是空白字符\n", var2 );
   }
   if( isspace(var3) )
   {
       printf("var3 = |%c| 是空白字符\n", var3 );
   }
   else
   {
       printf("var3 = |%c| 不是空白字符\n", var3 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |t| 不是空白字符
var2 = |1| 不是空白字符
var3 = | | 是空白字符
```

C 库函数 - isupper()

描述

C 库函数 **int isupper(int c)** 检查所传的字符是否是大写字母。

声明

下面是 isupper() 函数的声明。

```
int isupper(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个大写字母，则该函数返回非零值（true），否则返回 0（false）。

实例

下面的实例演示了 isupper() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   int var1 = 'M';
   int var2 = 'm';
   int var3 = '3';

   if( isupper(var1) )
   {
      printf("var1 = |%c| 是大写字母\n", var1 );
   }
   else
   {
      printf("var1 = |%c| 不是大写字母\n", var1 );
   }
   if( isupper(var2) )
   {
      printf("var2 = |%c| 是大写字母\n", var2 );
   }
   else
   {
      printf("var2 = |%c| 不是大写字母\n", var2 );
   }
   if( isupper(var3) )
   {
      printf("var3 = |%c| 是大写字母\n", var3 );
   }
   else
   {
      printf("var3 = |%c| 不是大写字母\n", var3 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |M| 是大写字母
var2 = |m| 不是大写字母
var3 = |3| 不是大写字母
```

C 库函数 - isxdigit()

描述

C 库函数 **int isxdigit(int c)**检查所传的字符是否是十六进制数字。

声明

下面是 isxdigit() 函数的声明。

```
int isxdigit(int c);
```

参数

- **c** -- 这是要检查的字符。

返回值

如果 c 是一个十六进制数字，则该函数返回非零值（true），否则返回 0（false）。

实例

下面的实例演示了 isxdigit() 函数的用法。

```
#include <stdio.h>
#include <ctype.h>

int main()
{
   char var1[] = "tuts";
   char var2[] = "0xE";

   if( isxdigit(var1[0]) )
   {
     printf("var1 = |%s| 是十六进制数字\n", var1 );
   }
   else
   {
     printf("var1 = |%s| 不是十六进制数字\n", var1 );
   }

   if( isxdigit(var2[0] ))
   {
     printf("var2 = |%s| 是十六进制数字\n", var2 );
   }
   else
   {
     printf("var2 = |%s| 不是十六进制数字\n", var2 );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
var1 = |tuts| 不是十六进制数字
var2 = |0xE| 是十六进制数字
```

C 标准库 - <errno.h>

简介

C 标准库的 **errno.h** 头文件定义了整数变量 **errno**，它是通过系统调用设置的，在错误事件中的某些库函数表明了什么发生了错误。该宏扩展为类型为 int 的可更改的左值，因此它可以被一个程序读取和修改。

在程序启动时，**errno** 设置为零，C 标准库中的特定函数修改它的值为一些非零值以表示某些类型的错误。您也可以在适当的时候修改它的值或重置为零。

errno.h 头文件也顶了以一系列表示不同错误代码的宏，这些宏应扩展为类型为 **int** 的整数常量表达式。

库宏

下面列出了头文件 errno.h 中定义的宏：

宏	描述
extern int errno	这是通过系统调用设置的宏，在错误事件中的某些库函数表明了什么发生了错误。
EDOM Domain Error	这个宏表示一个域错误，它在输入参数超出数学函数定义的域时发生，errno 被设置为 EDOM。
ERANGE Range Error	这个宏表示一个范围错误，它在输入参数超出数学函数定义的范围时发生，errno 被设置为 ERANGE。

C 库宏 - errno

描述

C 库宏 **extern int errno** 是通过系统调用设置的，在错误事件中的某些库函数表明了什么发生了错误。

声明

下面是 errno 宏的声明。

```
extern int errno
```

参数

- **NA**

返回值

- **NA**

实例

下面的实例演示了 errno 宏的用法。

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno ;

int main ()
{
   FILE *fp;

   fp = fopen("file.txt", "r");
   if( fp == NULL )
   {
      fprintf(stderr, "Value of errno: %d\n", errno);
      fprintf(stderr, "Error opening file: %s\n", strerror(errno));
   }
   else
   {
      fclose(fp);
   }

   return(0);
}
```

让我们编译并运行上面的程序，当文件 **file.txt** 不存在时，将产生以下结果：

```
Value of errno: 2
Error opening file: No such file or directory
```

C 库宏 - EDOM

描述

C 库宏 **EDOM** 表示一个域错误，它在输入参数超出数学函数定义的域时发生，errno 被设置为 EDOM。

声明

下面是 EDOM 宏的声明。

```
#define EDOM some_value
```

参数

- **NA**

返回值

- **NA**

实例

下面的实例演示了 EDOM 宏的用法。

```
#include <stdio.h>
#include <errno.h>
#include <math.h>

int main()
{
   double val;

   errno = 0;
   val = sqrt(-10);
   if(errno == EDOM)
   {
      printf("Invalid value \n");
   }
   else
   {
      printf("Valid value\n");
   }

   errno = 0;
   val = sqrt(10);
   if(errno == EDOM)
   {
      printf("Invalid value\n");
   }
   else
   {
      printf("Valid value\n");
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Invalid value
Valid value
```

C 库宏 - ERANGE

描述

C 库宏 **ERANGE** 表示一个范围错误，它在输入参数超出数学函数定义的范围时发生，errno 被设置为 ERANGE。

声明

下面是 ERANGE 宏的声明。

```
#define ERANGE some_value
```

参数

- **NA**

返回值

- **NA**

实例

下面的实例演示了 ERANGE 宏的用法。

```
#include <stdio.h>
#include <errno.h>
#include <math.h>

int main()
{
   double x;
   double value;

   x = 1.000000;
   value = log(x);
   if( errno == ERANGE )
   {
      printf("Log(%f) is out of range\n", x);
   }
   else
   {
      printf("Log(%f) = %f\n", x, value);
   }

   x = 0.000000;
   value = log(x);
   if( errno == ERANGE )
   {
      printf("Log(%f) is out of range\n" x);
   }
   else
   {
      printf("Log(%f) = %f\n", x, value);
   }

   return 0;
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Log(1.000000) = 1.609438
Log(0.000000) is out of range
```

C 标准库 - <float.h>

简介

C 标准库的 **float.h** 头文件包含了一组与浮点值相关的依赖于平台的常量。这些常量是由 ANSI C 提出的，这让程序更具有可移植性。在讲解这些常量之前，最好先弄清楚浮点数是由下面四个元素组成的：

组件	组件描述
S	符号 (+/-)
b	指数表示的基数，2 表示二进制，10 表示十进制，16 表示十六进制，等等...
e	指数，一个介于最小值 e_{min} 和最大值 e_{max} 之间的整数。
p	精度，基数 b 的有效位数

基于以上 4 个组成部分，一个浮点数的值如下：

```
floating-point = ( S ) p x b<sup>e</sup>

或

floating-point = (+/-) precision x base<sup>exponent</sup>
```

库宏

下面的值是特定实现的，且是通过 #define 指令来定义的，这些值都不得低于下边所给出的值。请注意，所有的实例 FLT 是指类型 float，DBL 是指类型 double，LDBL 是指类型 long double。

宏	描述
FLT_ROUNDS	定义浮点加法的舍入模式，它可以是下列任何一个值： -1 - 无法确定 0 - 趋向于零 1 - 去最近的值 2 - 趋向于正无穷 3 - 趋向于负无穷
FLT_RADIX 2	这个宏定义了指数表示的基数。基数 2 表示二进制，基数 10 表示十进制，基数 16 表示十六进制。
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	这些宏定义了 FLT_RADIX 基数中的位数。
FLT_DIG 6 DBL_DIG 10 LDBL_DIG 10	这些宏定义了舍入后不会改变表示的十进制数字的最大值（基数 10）。
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	这些宏定义了基数为 FLT_RADIX 时的指数的最小负整数值。
FLT_MIN_10_EXP -37 DBL_MIN_10_EXP -37 LDBL_MIN_10_EXP -37	这些宏定义了基数为 10 时的指数的最小负整数值。
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	这些宏定义了基数为 FLT_RADIX 时的指数的最大整数值。
FLT_MAX_10_EXP +37 DBL_MAX_10_EXP +37 LDBL_MAX_10_EXP +37	这些宏定义了基数为 10 时的指数的最大整数值。
FLT_MAX 1E+37 DBL_MAX 1E+37 LDBL_MAX 1E+37	这些宏定义最大的有限浮点值。
FLT_EPSILON 1E-5 DBL_EPSILON 1E-9 LDBL_EPSILON 1E-9	这些宏定义了可表示的最小有效数字。
FLT_MIN 1E-37 DBL_MIN 1E-37 LDBL_MIN 1E-37	这些宏定义了最小的浮点值。

实例

下面的实例演示了 float.h 文件中定义的一些常量的使用。

```
#include <stdio.h>
#include <float.h>

int main()
{
   printf("The maximum value of float = %.10e\n", FLT_MAX);
   printf("The minimum value of float = %.10e\n", FLT_MIN);

   printf("The number of digits in the number = %.10e\n", FLT_MANT_DIG);
}
```

让我们编译和运行上面的程序，这将产生下列结果：

```
The maximum value of float = 3.4028234664e+38
The minimum value of float = 1.1754943508e-38
The number of digits in the number = 7.2996655210e-312
```

C 标准库 - <limits.h>

简介

limits.h 头文件决定了各种变量类型的各种属性。定义在该头文件中的宏限制了各种变量类型（比如 char、int 和 long）的值。

这些限制指定了变量不能存储任何超出这些限制的值，例如一个无符号可以存储的最大值是 255。

库宏

下面的值是特定实现的，且是通过 #define 指令来定义的，这些值都不得低于下边所给出的值。

宏	值	描述
CHAR_BIT	8	定义一个字节的比特数。
SCHAR_MIN	-128	定义一个有符号字符的最小值。
SCHAR_MAX	127	定义一个有符号字符的最大值。
UCHAR_MAX	255	定义一个无符号字符的最大值。
CHAR_MIN	0	定义类型 char 的最小值，如果 char 表示负值，则它的值等于 SCHAR_MIN，否则等于 0。
CHAR_MAX	127	定义类型 char 的最大值，如果 char 表示负值，则它的值等于 SCHAR_MAX，否则等于 UCHAR_MAX。
MB_LEN_MAX	1	定义多字节字符中的最大字节数。
SHRT_MIN	-32768	定义一个短整型的最小值。
SHRT_MAX	+32767	定义一个短整型的最大值。
USHRT_MAX	65535	定义一个无符号短整型的最大值。
INT_MIN	-32768	定义一个整型的最小值。
INT_MAX	+32767	定义一个整型的最大值。
UINT_MAX	65535	定义一个无符号整型的最大值。
LONG_MIN	-2147483648	定义一个长整型的最小值。
LONG_MAX	+2147483647	定义一个长整型的最大值。
ULONG_MAX	4294967295	定义一个无符号长整型的最大值。

实例

下面的实例演示了 limit.h 文件中定义的一些常量的使用。

```
#include <stdio.h>
#include <limits.h>

int main()
{

   printf("The number of bits in a byte %d\n", CHAR_BIT);

   printf("The minimum value of SIGNED CHAR = %d\n", SCHAR_MIN);
   printf("The maximum value of SIGNED CHAR = %d\n", SCHAR_MAX);
   printf("The maximum value of UNSIGNED CHAR = %d\n", UCHAR_MAX);

   printf("The minimum value of SHORT INT = %d\n", SHRT_MIN);
   printf("The maximum value of SHORT INT = %d\n", SHRT_MAX);

   printf("The minimum value of INT = %d\n", INT_MIN);
   printf("The maximum value of INT = %d\n", INT_MAX);

   printf("The minimum value of CHAR = %d\n", CHAR_MIN);
   printf("The maximum value of CHAR = %d\n", CHAR_MAX);

   printf("The minimum value of LONG = %ld\n", LONG_MIN);
   printf("The maximum value of LONG = %ld\n", LONG_MAX);

   return(0);
}
```

让我们编译和运行上面的程序，这将产生下列结果：

```
The number of bits in a byte 8
The minimum value of SIGNED CHAR = -128
The maximum value of SIGNED CHAR = 127
The maximum value of UNSIGNED CHAR = 255
The minimum value of SHORT INT = -32768
The maximum value of SHORT INT = 32767
The minimum value of INT = -32768
The maximum value of INT = 32767
The minimum value of CHAR = -128
The maximum value of CHAR = 127
The minimum value of LONG = -2147483648
The maximum value of LONG = 2147483647
```

C 标准库 - <locale.h>

简介

locale.h 头文件定义了特定地域的设置，比如日期格式和货币符号。接下来我们将介绍一些宏，以及一个重要的结构 **struct lconv** 和两个重要的函数。

库宏

下面列出了头文件 locale.h 中定义的宏，这些宏将在下列的两个函数中使用：

宏	描述
LC_ALL	设置下面的所有选项。
LC_COLLATE	影响 strcoll 和 strxfrm 函数。
LC_CTYPE	影响所有字符函数。
LC_MONETARY	影响 localeconv 函数提供的货币信息。
LC_NUMERIC	影响 localeconv 函数提供的小数点格式化和信息。
LC_TIME	影响 strftime 函数。

库函数

下面列出了头文件 locale.h 中定义的函数：

函数	描述
char *setlocale(int category, const char *locale)	设置或读取地域化信息。
struct lconv *localeconv(void)	设置或读取地域化信息。

库结构

```
typedef struct {
   char *decimal_point;
   char *thousands_sep;
   char *grouping;
   char *int_curr_symbol;
   char *currency_symbol;
   char *mon_decimal_point;
   char *mon_thousands_sep;
   char *mon_grouping;
   char *positive_sign;
   char *negative_sign;
   char int_frac_digits;
   char frac_digits;
   char p_cs_precedes;
   char p_sep_by_space;
   char n_cs_precedes;
   char n_sep_by_space;
   char p_sign_posn;
   char n_sign_posn;
} lconv
```

以下是各字段的描述：

字段	描述
decimal_point	用于非货币值的小数点字符。
thousands_sep	用于非货币值的千位分隔符。
grouping	一个表示非货币量中每组数字大小的字符串。每个字符代表一个整数值，每个整数指定当前组的位数。值为 0 意味着前一个值将应用于剩余的分组。
int_curr_symbol	国际货币符号使用的字符串。前三个字符是由 ISO 4217:1987 指定的，第四个字符用于分隔货币符号和货币量。
currency_symbol	用于货币的本地符号。
mon_decimal_point	用于货币值的小数点字符。
mon_thousands_sep	用于货币值的千位分隔符。
mon_grouping	一个表示货币值中每组数字大小的字符串。每个字符代表一个整数值，每个整数指定当前组的位数。值为 0 意味着前一个值将应用于剩余的分组。
positive_sign	用于正货币值的字符。
negative_sign	用于负货币值的字符。
int_frac_digits	国际货币值中小数点后要显示的位数。
frac_digits	货币值中小数点后要显示的位数。
p_cs_precedes	如果等于 1，则 currency_symbol 出现在正货币值之前。如果等于 0，则 currency_symbol 出现在正货币值之后。
p_sep_by_space	如果等于 1，则 currency_symbol 和正货币值之间使用空格分隔。如果等于 0，则 currency_symbol 和正货币值之间不使用空格分隔。
n_cs_precedes	如果等于 1，则 currency_symbol 出现在负货币值之前。如果等于 0，则 currency_symbol 出现在负货币值之后。
n_sep_by_space	如果等于 1，则 currency_symbol 和负货币值之间使用空格分隔。如果等于 0，则 currency_symbol 和负货币值之间不使用空格分隔。
p_sign_posn	表示正货币值中正号的位置。
n_sign_posn	表示负货币值中负号的位置。

下面的值用于 **p_sign_posn** 和 **n_sign_posn**:

值	描述
0	封装值和 currency_symbol 的括号。
1	放置在值和 currency_symbol 之前的符号。
2	放置在值和 currency_symbol 之后的符号。
3	紧挨着放置在值和 currency_symbol 之前的符号。
4	紧挨着放置在值和 currency_symbol 之后的符号。

C 库函数 - setlocale()

描述

C 库函数 **char *setlocale(int category, const char* locale)** 设置或读取地域化信息。

声明

下面是 setlocale() 函数的声明。

```
char *setlocale(int category, const char *locale)
```

参数

- **category** -- 这是一个已命名的常量，指定了受区域设置影响的函数类别。
  - **LC_ALL** 包括下面的所有选项。
  - **LC_COLLATE** 字符串比较。参见 strcoll()。
  - **LC_CTYPE** 字符分类和转换。例如 strtoupper()。
  - **LC_MONETARY** 货币格式，针对 localeconv()。
  - **LC_NUMERIC** 小数点分隔符，针对 localeconv()。
  - **LC_TIME** 日期和时间格式，针对 strftime()。
  - **LC_MESSAGES** 系统响应。
- **locale** -- 如果 locale 是 NULL 或空字符串 ""，则区域名称将根据环境变量值来设置，其名称与上述的类别名称相同。

返回值

如果成功调用 setlocale()，则返回一个对应于区域设置的不透明的字符串。如果请求无效，则返回值是 NULL。

实例

下面的实例演示了 setlocale() 函数的用法。

```
#include <locale.h>
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t currtime;
   struct tm *timer;
   char buffer[80];

   time( &currtime );
   timer = localtime( &currtime );

   printf("Locale is: %s\n", setlocale(LC_ALL, "en_GB"));
   strftime(buffer,80,"%c", timer );
   printf("Date is: %s\n", buffer);

   printf("Locale is: %s\n", setlocale(LC_ALL, "de_DE"));
   strftime(buffer,80,"%c", timer );
   printf("Date is: %s\n", buffer);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Locale is: en_GB
Date is: Thu 23 Aug 2012 06:39:32 MST
Locale is: de_DE
Date is: Do 23 Aug 2012 06:39:32 MST
```

C 库函数 - localeconv()

描述

C 库函数 **struct lconv *localeconv(void)** 设置或读取地域化信息。它会返回一个 **lconv** 结构类型的对象。

声明

下面是 localeconv() 函数的声明。

```
struct lconv *localeconv(void)
```

参数

- **NA**

返回值

该函数返回一个指向当前区域 **struct lconv** 的指针，它的结构如下：

```
typedef struct {
   char *decimal_point;
   char *thousands_sep;
   char *grouping;
   char *int_curr_symbol;
   char *currency_symbol;
   char *mon_decimal_point;
   char *mon_thousands_sep;
   char *mon_grouping;
   char *positive_sign;
   char *negative_sign;
   char int_frac_digits;
   char frac_digits;
   char p_cs_precedes;
   char p_sep_by_space;
   char n_cs_precedes;
   char n_sep_by_space;
   char p_sign_posn;
   char n_sign_posn;
} lconv
```

实例

下面的实例演示了 localeconv() 函数的用法。

```
#include <locale.h>
#include <stdio.h>

int main ()
{
   struct lconv * lc;

   setlocale(LC_MONETARY, "it_IT");
   lc = localeconv();
   printf("Local Currency Symbol: %s\n",lc->currency_symbol);
   printf("International Currency Symbol: %s\n",lc->int_curr_symbol);

   setlocale(LC_MONETARY, "en_US");
   lc = localeconv();
   printf("Local Currency Symbol: %s\n",lc->currency_symbol);
   printf("International Currency Symbol: %s\n",lc->int_curr_symbol);

   setlocale(LC_MONETARY, "en_GB");
   lc = localeconv();
   printf ("Local Currency Symbol: %s\n",lc->currency_symbol);
   printf ("International Currency Symbol: %s\n",lc->int_curr_symbol);

   printf("Decimal Point = %s\n", lc->decimal_point);

   return 0;
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Local Currency Symbol: EUR
International Currency Symbol: EUR
Local Currency Symbol: $
International Currency Symbol: USD
Local Currency Symbol: £
International Currency Symbol: GBP
Decimal Point = .
```

C 标准库 - <math.h>

简介

math.h 头文件定义了各种数学函数和一个宏。在这个库中所有可用的功能都带有一个 **double** 类型的参数，且都返回 **double** 类型的结果。

库宏

下面是这个库中定义的唯一的一个宏：

宏	描述
HUGE_VAL	当函数的结果不可以表示为浮点数时。如果是因为结果的幅度太大以致于无法表示，则函数会设置 errno 为 ERANGE 来表示范围错误，并返回一个由宏 HUGE_VAL 或者它的否定（- HUGE_VAL）命名的一个特定的很大的值。 如果结果的幅度太小，则会返回零值。在这种情况下，error 可能会被设置为 ERANGE，也有可能不会被设置为 ERANGE。

库函数

下面列出了头文件 math.h 中定义的函数：

函数	描述
double acos(double x)	返回以弧度表示的 x 的反余弦。
double asin(double x)	返回以弧度表示的 x 的反正弦。
double atan(double x)	返回以弧度表示的 x 的反正切。
double atan2(double y, double x)	返回以弧度表示的 y/x 的反正切。y 和 x 的值的符号决定了正确的象限。
double cos(double x)	返回弧度角 x 的余弦。
double cosh(double x)	返回 x 的双曲余弦。
double sin(double x)	返回弧度角 x 的正弦。

double sinh(double x)	返回 x 的双曲正弦。
double tanh(double x)	返回 x 的双曲正切。
double exp(double x)	返回 e 的 x 次幂的值。
double frexp(double x, int *exponent)	把浮点数 x 分解成尾数和指数。返回值是尾数，并将指数存入 exponent 中。所得的值是 x = mantissa * 2 ^ exponent。
double ldexp(double x, int exponent)	返回 x 乘以 2 的 exponent 次幂。
double log(double x)	返回 x 的自然对数（基数为 e 的对数）。
double log10(double x)	返回 x 的常用对数（基数为 10 的对数）。
double modf(double x, double *integer)	返回值为小数部分（小数点后的部分），并设置 integer 为整数部分。
double pow(double x, double y)	返回 x 的 y 次幂。
double sqrt(double x)	返回 x 的平方根。
double ceil(double x)	返回大于或等于 x 的最小的整数值。
double fabs(double x)	返回 x 的绝对值。
double floor(double x)	返回小于或等于 x 的最大的整数值。
double fmod(double x, double y)	返回 x 除以 y 的余数。

C 库函数 - acos()

描述

C 库函数 **double acos(double x)** 返回以弧度表示的 **x** 的反余弦。

声明

下面是 acos() 函数的声明。

```
double acos(double x)
```

参数

- **x** -- 介于 [-1,+1] 区间的浮点值。

返回值

该函数返回以弧度表示的 x 的反余弦，弧度区间为 [0, pi]。

实例

下面的实例演示了 acos() 函数的用法。

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main ()
{
   double x, ret, val;

   x = 0.9;
   val = 180.0 / PI;

   ret = acos(x) * val;
   printf("%lf 的反余弦是 %lf 度", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
0.900000 的反余弦是 25.855040 度
```

C 库函数 - asin()

描述

C 库函数 **double asin(double x)** 返回以弧度表示的 **x** 的反正弦。

声明

下面是 asin() 函数的声明。

```
double asin(double x)
```

参数

- **x** -- 介于 [-1,+1] 区间的浮点值。

返回值

该函数返回以弧度表示的 x 的反正弦，弧度区间为 [-pi/2,+pi/2]。

实例

下面的实例演示了 asin() 函数的用法。

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main ()
{
   double x, ret, val;
   x = 0.9;
   val = 180.0 / PI;

   ret = asin(x) * val;
   printf("%lf 的反正弦是 %lf 度", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
0.900000 的反正弦是 64.190609 度
```

C 库函数 - atan()

描述

C 库函数 **double atan(double x)** 返回以弧度表示的 **x** 的反正切。

声明

下面是 atan() 函数的声明。

```
double atan(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回以弧度表示的 x 的反正切，弧度区间为 [-pi/2,+pi/2]。

实例

下面的实例演示了 atan() 函数的用法。

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main ()
{
   double x, ret, val;
   x = 1.0;
   val = 180.0 / PI;

   ret = atan (x) * val;
   printf("%lf 的反正切是 %lf 度", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
1.000000 的反正切是 45.000000 度
```

C 库函数 - atan2()

描述

C 库函数 **double atan2(doubly y, double x)** 返回以弧度表示的 **y/x** 的反正切。y 和 x 的值的符号决定了正确的象限。

声明

下面是 atan2() 函数的声明。

```
double atan2(doubly y, double x)
```

参数

- **x** -- 代表 x 轴坐标的浮点值。
- **y** -- 代表 y 轴坐标的浮点值。

返回值

该函数返回以弧度表示的 y/x 的反正切，弧度区间为 [-pi,+pi]。

实例

下面的实例演示了 atan2() 函数的用法。

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main ()
{
   double x, y, ret, val;

   x = -7.0;
   y = 7.0;
   val = 180.0 / PI;

   ret = atan2 (y,x) * val;
   printf("x = %lf, y = %lf 的反正切", x, y);
   printf("是 %lf 度\n", ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
x = -7.000000, y = 7.000000 的反正切是 135.000000 度
```

C 库函数 - cos()

描述

C 库函数 **double cos(double x)** 返回弧度角 **x** 的余弦。

声明

下面是 cos() 函数的声明。

```
double cos(double x)
```

参数

- **x** -- 浮点值，代表了一个以弧度表示的角度。

返回值

该函数返回 x 的余弦。

实例

下面的实例演示了 cos() 函数的用法。

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main ()
{
   double x, ret, val;

   x = 60.0;
   val = PI / 180.0;
   ret = cos( x*val );
   printf("%lf 的余弦是 %lf 度\n", x, ret);

   x = 90.0;
   val = PI / 180.0;
   ret = cos( x*val );
   printf("%lf 的余弦是 %lf 度\n", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
60.000000 的余弦是 0.500000 度
90.000000 的余弦是 0.000000 度
```

C 库函数 - cosh()

描述

C 库函数 **double cosh(double x)** 返回 **x** 的双曲余弦。

声明

下面是 cosh() 函数的声明。

```
double cosh(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的双曲余弦。

实例

下面的实例演示了 cosh() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x;

   x = 0.5;
   printf("%lf 的双曲余弦是 %lf\n", x, cosh(x));

   x = 1.0;
   printf("%lf 的双曲余弦是 %lf\n", x, cosh(x));

   x = 1.5;
   printf("%lf 的双曲余弦是 %lf\n", x, cosh(x));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
0.500000 的双曲余弦是 1.127626
1.000000 的双曲余弦是 1.543081
1.500000 的双曲余弦是 2.352410
```

C 库函数 - sin()

描述

C 库函数 **double sin(double x)** 返回弧度角 **x** 的正弦。

声明

下面是 sin() 函数的声明。

```
double sin(double x)
```

参数

- **x** -- 浮点值，代表了一个以弧度表示的角度。

返回值

该函数返回 x 的正弦。

实例

下面的实例演示了 sin() 函数的用法。

```
#include <stdio.h>
#include <math.h>

#define PI 3.14159265

int main ()
{
   double x, ret, val;

   x = 45.0;
   val = PI / 180;
   ret = sin(x*val);
   printf("%lf 的正弦是 %lf 度", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
45.000000 的正弦是 0.707107 度
```

C 库函数 - sinh()

描述

C 库函数 **double sinh(double x)** 返回 **x** 的双曲正弦。

声明

下面是 sinh() 函数的声明。

```
double sinh(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的双曲正弦。

实例

下面的实例演示了 sinh() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x, ret;
   x = 0.5;

   ret = sinh(x);
   printf("%lf 的双曲正弦是 %lf 度", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
0.500000 的双曲正弦是 0.521095 度
```

C 库函数 - tanh()

描述

C 库函数 **double tanh(double x)** 返回 **x** 的双曲正切。

声明

下面是 tanh() 函数的声明。

```
double tanh(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的双曲正切。

实例

下面的实例演示了 tanh() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x, ret;
   x = 0.5;

   ret = tanh(x);
   printf("%lf 的双曲正切是 %lf 度", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
0.500000 的双曲正切是 0.462117 度
```

C 库函数 - exp()

描述

C 库函数 **double exp(double x)** 返回 **e** 的 **x** 次幂的值。

声明

下面是 exp() 函数的声明。

```
double exp(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 e 的 x 次幂。

实例

下面的实例演示了 exp() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x = 0;

   printf("e 的 %lf 次幂是 %lf\n", x, exp(x));
   printf("e 的 %lf 次幂是 %lf\n", x+1, exp(x+1));
   printf("e 的 %lf 次幂是 %lf\n", x+2, exp(x+2));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
e 的 0.000000 次幂是 1.000000
e 的 1.000000 次幂是 2.718282
e 的 2.000000 次幂是 7.389056
```

C 库函数 - frexp()

描述

C 库函数 **double frexp(double x, int *exponent)** 把浮点数 x 分解成尾数和指数。返回值是尾数，并将指数存入 **exponent** 中。所得的值是 **x = mantissa * 2 ^ exponent**。

声明

下面是 frexp() 函数的声明。

```
double frexp(double x, int *exponent)
```

参数

- **x** -- 要被计算的浮点值。
- **exponent** -- 指向一个对象的指针，该对象存储了指数的值。

返回值

该函数返回规格化小数。如果参数 x 不为零，则规格化小数是 x 的二次方，且它的绝对值范围从 1/2（包含）到 1（不包含）。如果 x 为零，则规格化小数是零，且零存储在 exp 中。

实例

下面的实例演示了 frexp() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x = 1024, fraction;
   int e;

   fraction = frexp(x, &e);
   printf("x = %.2lf = %.2lf * 2^%d\n", x, fraction, e);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
x = 1024.00 = 0.50 * 2^11
```

C 库函数 - ldexp()

描述

C 库函数 **double ldexp(double x, int exponent)** 返回 **x** 乘以 2 的 **exponent** 次幂。

声明

下面是 ldexp() 函数的声明。

```
double ldexp(double x, int exponent)
```

参数

- **x** -- 代表有效位数的浮点值。
- **exponent** -- 指数的值。

返回值

该函数返回 x * 2 ^{exp}。

实例

下面的实例演示了 ldexp() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x, ret;
   int n;

   x = 0.65;
   n = 3;
   ret = ldexp(x ,n);
   printf("%f * 2^%d = %f\n", x, n, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
0.650000 * 2^3 = 5.200000
```

C 库函数 - log()

描述

C 库函数 **double log(double x)** 返回 **x** 的自然对数（基数为 e 的对数）。

声明

下面是 log() 函数的声明。

```
double log(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的自然对数。

实例

下面的实例演示了 log() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x, ret;
   x = 2.7;

   /* 计算 log(2.7) */
   ret = log(x);
   printf("log(%lf) = %lf", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
log(2.700000) = 0.993252
```

C 库函数 - log10()

描述

C 库函数 **double log10(double x)** 返回 **x** 的常用对数（基数为 10 的对数）。

声明

下面是 log10() 函数的声明。

```
double log10(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的常用对数，x 的值大于 0。

实例

下面的实例演示了 log10() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   double x, ret;
   x = 10000;

   /* 计算 log10(10000) */
   ret = log10(x);
   printf("log10(%lf) = %lf\n", x, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
log10(10000.000000) = 4.000000
```

C 库函数 - modf()

描述

C 库函数 **double modf(double x, double *integer)** 返回值为小数部分（小数点后的部分），并设置 integer 为整数部分。

声明

下面是 modf() 函数的声明。

```
double modf(double x, double *integer)
```

参数

- **x** -- 浮点值。
- **integer** -- 指向一个对象的指针，该对象存储了整数部分。

返回值

该函数返回 x 的小数部分，符号与 x 相同。

实例

下面的实例演示了 modf() 函数的用法。

```
#include<stdio.h>
#include<math.h>

int main ()
{
   double x, fractpart, intpart;

   x = 8.123456;
   fractpart = modf(x, &intpart);

   printf("整数部分 = %lf\n", intpart);
   printf("小数部分 = %lf \n", fractpart);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
整数部分 = 8.000000
小数部分 = 0.123456
```

C 库函数 - pow()

描述

C 库函数 **double pow(double x, double y)** 返回 **x** 的 **y** 次幂，即 x^y。

声明

下面是 pow() 函数的声明。

```
double pow(double x, double y)
```

参数

- **x** -- 代表基数的浮点值。
- **y** -- 代表指数的浮点值。

返回值

该函数返回 x 的 y 次幂的结果。

实例

下面的实例演示了 pow() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   printf("值 8.0 ^ 3 = %lf\n", pow(8.0, 3));

   printf("值 3.05 ^ 1.98 = %lf", pow(3.05, 1.98));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
值 8.0 ^ 3 = 512.000000
值 3.05 ^ 1.98 = 9.097324
```

C 库函数 - sqrt()

描述

C 库函数 **double sqrt(double x)** 返回 **x** 的平方根。

声明

下面是 sqrt() 函数的声明。

```
double sqrt(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的平方根。

实例

下面的实例演示了 sqrt() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{

   printf("%lf 的平方根是 %lf\n", 4.0, sqrt(4.0) );
   printf("%lf 的平方根是 %lf\n", 5.0, sqrt(5.0) );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
4.000000 的平方根是 2.000000
5.000000 的平方根是 2.236068
```

C 库函数 - ceil()

描述

C 库函数 **double ceil(double x)** 返回大于或等于 **x** 的最小的整数值。

声明

下面是 ceil() 函数的声明。

```
double ceil(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回不小于 x 的最小整数值。

实例

下面的实例演示了 ceil() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   float val1, val2, val3, val4;

   val1 = 1.6;
   val2 = 1.2;
   val3 = 2.8;
   val4 = 2.3;

   printf ("value1 = %.1lf\n", ceil(val1));
   printf ("value2 = %.1lf\n", ceil(val2));
   printf ("value3 = %.1lf\n", ceil(val3));
   printf ("value4 = %.1lf\n", ceil(val4));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
value1 = 2.0
value2 = 2.0
value3 = 3.0
value4 = 3.0
```

C 库函数 - fabs()

描述

C 库函数 **double fabs(double x)** 返回 **x** 的绝对值。

声明

下面是 fabs() 函数的声明。

```
double fabs(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回 x 的绝对值。

实例

下面的实例演示了 fabs() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   int a, b;
   a = 1234;
   b = -344;

   printf("%d 的绝对值是 %lf\n", a, fabs(a));
   printf("%d 的绝对值是 %lf\n", b, fabs(b));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
1234 的绝对值是 1234.000000
-344 的绝对值是 344.000000
```

C 库函数 - floor()

描述

C 库函数 **double floor(double x)** 返回小于或等于 **x** 的最大的整数值。

声明

下面是 floor() 函数的声明。

```
double floor(double x)
```

参数

- **x** -- 浮点值。

返回值

该函数返回不大于 x 的最大整数值。

实例

下面的实例演示了 floor() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   float val1, val2, val3, val4;

   val1 = 1.6;
   val2 = 1.2;
   val3 = 2.8;
   val4 = 2.3;

   printf("Value1 = %.1lf\n", floor(val1));
   printf("Value2 = %.1lf\n", floor(val2));
   printf("Value3 = %.1lf\n", floor(val3));
   printf("Value4 = %.1lf\n", floor(val4));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Value1 = 1.0
Value2 = 1.0
Value3 = 2.0
Value4 = 2.0
```

C 库函数 - fmod()

描述

C 库函数 **double fmod(double x, double y)** 返回 **x** 除以 **y** 的余数。

声明

下面是 fmod() 函数的声明。

```
double fmod(double x, double y)
```

参数

- **x** -- 代表分子的浮点值。
- **y** -- 代表分母的浮点值。

返回值

该函数返回 x/y 的余数。

实例

下面的实例演示了 fmod() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main ()
{
   float a, b;
   int c;
   a = 9.2;
   b = 3.7;
   c = 2;
   printf("%f / %d 的余数是 %lf\n", a, c, fmod(a,c));
   printf("%f / %f 的余数是 %lf\n", a, b, fmod(a,b));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
9.200000 / 2 的余数是 1.200000
9.200000 / 3.700000 的余数是 1.800000
```

C 标准库 - <setjmp.h>

简介

setjmp.h 头文件定义了宏 **setjmp()**、函数 **longjmp()** 和变量类型 **jmp_buf**，该变量类型会绕过正常的函数调用和返回规则。

库变量

下面列出了头文件 setjmp.h 中定义的变量：

变量	描述
jmp_buf	这是一个用于存储宏 **setjmp()** 和函数 **longjmp()** 相关信息的数组类型。

库宏

下面是这个库中定义的唯一的一个宏：

宏	描述
int setjmp(jmp_buf environment)	这个宏把当前环境保存在变量 **environment** 中，以便函数 **longjmp()** 后续使用。如果这个宏直接从宏调用中返回，则它会返回零，但是如果它从 **longjmp()** 函数调用中返回，则它会返回一个非零值。

库函数

下面是头文件 setjmp.h 中定义的唯一的一个函数：

函数	描述
void longjmp(jmp_buf environment, int value)	该函数恢复最近一次调用 **setjmp()** 宏时保存的环境，**jmp_buf** 参数的设置是由之前调用 setjmp() 生成的。

C 库宏 - setjmp()

描述

C 库宏 **int setjmp(jmp_buf environment)** 把当前环境保存在变量 environment 中，以便函数 **longjmp()** 后续使用。如果这个宏直接从宏调用中返回，则它会返回零，但是如果它从 **longjmp()** 函数调用中返回，则它会返回一个传给 longjmp 作为第二个参数的非零值。

声明

下面是 setjmp() 宏的声明。

```
int setjmp(jmp_buf environment)
```

参数

- **environment** -- 这是一个类型为 jmp_buf 的用于存储环境信息的对象。

返回值

这个宏可能不只返回一次。第一次，在直接调用它时，它总是返回零。当调用 longjmp 时带有设置的环境信息，这个宏会再次返回，此时它返回的值会传给 longjmp 作为第二个参数。

实例

下面的实例演示了 setjmp() 宏的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

int main()
{
   int val;
   jmp_buf env_buffer;

   /* 保存 longjmp 的调用环境 */
   val = setjmp( env_buffer );
   if( val != 0 )
   {
      printf("从 longjmp() 返回值 = %s\n", val);
      exit(0);
   }
   printf("跳转函数调用\n");
   jmpfunction( env_buffer );

   return(0);
}

void jmpfunction(jmp_buf env_buf)
{
   longjmp(env_buf, "w3cschool.cc");
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
跳转函数调用
从 longjmp() 返回值 = w3cschool.cc
```

C 库函数 - longjmp()

描述

C 库函数 **void longjmp(jmp_buf environment, int value)** 恢复最近一次调用 **setjmp()** 宏时保存的环境，**jmp_buf** 参数的设置是由之前调用 setjmp() 生成的。

声明

下面是 longjmp() 函数的声明。

```
void longjmp(jmp_buf environment, int value)
```

参数

- **environment** -- 这是一个类型为 **jmp_buf** 的对象，包含了调用 setjmp 时存储的环境信息。
- **value** -- 这是 **setjmp** 表达式要判断的值。

返回值

该函数不返回任何值。

实例

下面的实例演示了 longjmp() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

int main()
{
   int val;
   jmp_buf env_buffer;

   /* 保存 longjmp 的调用环境 */
   val = setjmp( env_buffer );
   if( val != 0 )
   {
      printf("从 longjmp() 返回值 = %s\n", val);
      exit(0);
   }
   printf("跳转函数调用\n");
   jmpfunction( env_buffer );

   return(0);
}

void jmpfunction(jmp_buf env_buf)
{
   longjmp(env_buf, "w3cschool.cc");
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
跳转函数调用
从 longjmp() 返回值 = w3cschool.cc
```

C 标准库 - <signal.h>

简介

signal.h 头文件定义了一个变量类型 **sig_atomic_t**、两个函数调用和一些宏来处理程序执行期间报告的不同信号。

库变量

下面是头文件 signal.h 中定义的变量类型：

变量	描述
sig_atomic_t	这是 **int** 类型，在信号处理程序中作为变量使用。它是一个对象的整数类型，该对象可以作为一个原子实体访问，即使存在异步信号时，该对象可以作为一个原子实体访问。

库宏

下面是头文件 signal.h 中定义的宏，这些宏将在下列两个函数中使用。**SIG_** 宏与 signal 函数一起使用来定义信号的功能。

宏	描述
SIG_DFL	默认的信号处理程序。
SIG_ERR	表示一个信号错误。
SIG_IGN	忽视信号。

SIG 宏用于表示以下各种条件的信号号码：

宏	描述
SIGABRT	程序异常终止。
SIGFPE	算术运算出错，如除数为 0 或溢出。
SIGILL	非法函数映象，如非法指令。
SIGINT	中断信号，如 ctrl-C。
SIGSEGV	非法访问存储器，如访问不存在的内存单元。
SIGTERM	发送给本程序的终止请求信号。

库函数

下面是头文件 signal.h 中定义的函数：

函数	描述
void (*signal(int sig, void (*func)(int)))(int)	这个函数设置一个函数来处理信号，即信号处理程序。
int raise(int sig)	这个函数会促使生成信号 **sig**。sig 参数与 SIG 宏兼容。

C 库函数 - signal()

描述

C 库函数 **void (*signal(int sig, void (*func)(int)))(int)** 设置一个函数来处理信号，即带有 **sig** 参数的信号处理程序。

声明

下面是 signal() 函数的声明。

```
void (*signal(int sig, void (*func)(int)))(int)
```

参数

- **sig** -- 在信号处理程序中作为变量使用的信号码。下面是一些重要的标准信号常量：

宏	信号
SIGABRT	(Signal Abort) 程序异常终止。
SIGFPE	(Signal Floating-Point Exception) 算术运算出错，如除数为 0 或溢出（不一定是浮点运算）。
SIGILL	(Signal Illegal Instruction) 非法函数映象，如非法指令，通常是由于代码中的某个变体或者尝试执行数据导致的。
SIGINT	(Signal Interrupt) 中断信号，如 ctrl-C，通常由用户生成。
SIGSEGV	(Signal Segmentation Violation) 非法访问存储器，如访问不存在的内存单元。
SIGTERM	(Signal Terminate) 发送给本程序的终止请求信号。

- **func** -- 一个指向函数的指针。它可以是一个由程序定义的函数，也可以是下面预定义函数之一：

SIG_DFL	默认的信号处理程序。
SIG_IGN	忽视信号。

返回值

该函数返回信号处理程序之前的值，当发生错误时返回 SIG_ERR。

实例

下面的实例演示了 signal() 函数的用法。

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void sighandler(int);

int main()
{
   signal(SIGINT, sighandler);

   while(1)
   {
      printf("开始休眠一秒钟...\n");
      sleep(1);
   }

   return(0);
}

void sighandler(int signum)
{
   printf("捕获信号 %d，跳出...\n", signum);
   exit(1);
}
```

让我们编译并运行上面的程序，这将产生以下结果，且程序会进入无限循环，需使用 CTRL + C 键跳出程序。

```
开始休眠一秒钟...
开始休眠一秒钟...
开始休眠一秒钟...
开始休眠一秒钟...
开始休眠一秒钟...
捕获信号 2，跳出...
```

C 库函数 - raise()

描述

C 库函数 **int raise(int sig)** 会促使生成信号 **sig**。**sig** 参数与 SIG 宏兼容。

声明

下面是 raise() 函数的声明。

```
int raise(int sig)
```

参数

- **sig** -- 要发送的信号码。下面是一些重要的标准信号常量：

宏	信号
SIGABRT	(Signal Abort) 程序异常终止。
SIGFPE	(Signal Floating-Point Exception) 算术运算出错，如除数为 0 或溢出（不一定是浮点运算）。
SIGILL	(Signal Illegal Instruction) 非法函数映象，如非法指令，通常是由于代码中的某个变体或者尝试执行数据导致的。
SIGINT	(Signal Interrupt) 中断信号，如 ctrl-C，通常由用户生成。
SIGSEGV	(Signal Segmentation Violation) 非法访问存储器，如访问不存在的内存单元。
SIGTERM	(Signal Terminate) 发送给本程序的终止请求信号。

返回值

如果成功该函数返回零，否则返回非零。

实例

下面的实例演示了 raise() 函数的用法。

```
#include <signal.h>
#include <stdio.h>

void signal_catchfunc(int);

int main()
{
    int ret;

   ret = signal(SIGINT, signal_catchfunc);

   if( ret == SIG_ERR)
   {
      printf("错误：不能设置信号处理程序。\n");
      exit(0);
   }
   printf("开始生成一个信号\n");
   ret = raise(SIGINT);
   if( ret !=0 )
   {
      printf("错误：不能生成 SIGINT 信号。\n");
      exit(0);
   }

   printf("退出...\n");
   return(0);
}

void signal_catchfunc(int signal)
{
   printf("!! 信号捕获 !!\n");
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
开始生成一个信号
!! 信号捕获 !!
退出...
```

C 标准库 - <stdarg.h>

简介

stdarg.h 头文件定义了一个变量类型 **va_list** 和三个宏，这三个宏可用于在参数个数未知（即参数个数可变）时获取函数中的参数。

可变参数的函数通在参数列表的末尾是使用省略号(,...)定义的。

库变量

下面是头文件 stdarg.h 中定义的变量类型：

变量	描述
va_list	这是一个适用于 **va_start()**、**va_arg()** 和 **va_end()** 这三个宏存储信息的类型。

库宏

下面是头文件 stdarg.h 中定义的宏：

宏	描述
void va_start(va_list ap, last_arg)	这个宏初始化 **ap** 变量，它与 **va_arg** 和 **va_end** 宏是一起使用的。**last_arg** 是最后一个传递给函数的已知的固定参数，即省略号之前的参数。
type va_arg(va_list ap, type)	这个宏检索函数参数列表中类型为 **type** 的下一个参数。
void va_end(va_list ap)	这个宏允许使用了 **va_start** 宏的带有可变参数的函数返回。如果在从函数返回之前没有调用 **va_end**，则结果为未定义。

C 库宏 - va_start()

描述

C 库宏 **void va_start(va_list ap, last_arg)** 初始化 **ap** 变量，它与 **va_arg** 和 **va_end** 宏是一起使用的。**last_arg** 是最后一个传递给函数的已知的固定参数，即省略号之前的参数。

这个宏必须在使用 **va_arg** 和 **va_end** 之前被调用。

声明

下面是 va_start() 宏的声明。

```
void va_start(va_list ap, last_arg);
```

参数

- **ap** -- 这是一个 **va_list** 类型的对象，它用来存储通过 **va_arg** 获取额外参数时所必需的信息。
- **last_arg** -- 最后一个传递给函数的已知的固定参数。

返回值

NA

实例

下面的实例演示了 va_start() 宏的用法。

```
#include<stdarg.h>
#include<stdio.h>

int sum(int, ...);

int main(void)
{
   printf("10、20 和 30 的和 = %d\n",  sum(3, 10, 20, 30) );
   printf("4、20、25 和 30 的和 = %d\n",  sum(4, 4, 20, 25, 30) );

   return 0;
}

int sum(int num_args, ...)
{
   int val = 0;
   va_list ap;
   int i;

   va_start(ap, num_args);
   for(i = 0; i < num_args; i++)
   {
      val += va_arg(ap, int);
   }
   va_end(ap);

   return val;
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
10、20 和 30 的和 = 60
4、20、25 和 30 的和 = 79
```

C 库宏 - va_arg()

描述

C 库宏 **type va_arg(va_list ap, type)** 检索函数参数列表中类型为 **type** 的下一个参数。它无法判断检索到的参数是否是传给函数的最后一个参数。

声明

下面是 va_arg() 宏的声明。

```
type va_arg(va_list ap, type)
```

参数

- **ap** -- 这是一个 **va_list** 类型的对象，存储了有关额外参数和检索状态的信息。该对象应在第一次调用 va_arg 之前通过调用 va_start 进行初始化。
- **type** -- 这是一个类型名称。该类型名称是作为扩展自该宏的表达式的类型来使用的。

返回值

该宏返回下一个额外的参数，是一个类型为 type 的表达式。

实例

下面的实例演示了 va_arg() 宏的用法。

```
#include <stdarg.h>
#include <stdio.h>

int sum(int, ...);

int main()
{
   printf("15 和 56 的和 = %d\n",  sum(2, 15, 56) );
   return 0;
}

int sum(int num_args, ...)
{
   int val = 0;
   va_list ap;
   int i;

   va_start(ap, num_args);
   for(i = 0; i < num_args; i++)
   {
      val += va_arg(ap, int);
   }
   va_end(ap);

   return val;
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
15 和 56 的和 = 71
```

C 库宏 - va_end()

描述

C 库宏 **void va_end(va_list ap)** 允许使用了 **va_start** 宏的带有可变参数的函数返回。如果在从函数返回之前没有调用 **va_end**，则结果为未定义。

声明

下面是 va_end() 宏的声明。

```
void va_end(va_list ap)
```

参数

- **ap** -- 这是之前由同一函数中的 va_start 初始化的 va_list 对象。

返回值

该宏不返回任何值。

实例

下面的实例演示了 va_end() 宏的用法。

```
#include <stdarg.h>
#include <stdio.h>

int mul(int, ...);

int main()
{
   printf("15 * 12 = %d\n",  mul(2, 15, 12) );

   return 0;
}

int mul(int num_args, ...)
{
   int val = 1;
   va_list ap;
   int i;

   va_start(ap, num_args);
   for(i = 0; i < num_args; i++)
   {
      val *= va_arg(ap, int);
   }
   va_end(ap);

   return val;
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
15 * 12 =  180
```

C 标准库 - <stddef.h>

简介

stddef .h 头文件定义了各种变量类型和宏。这些定义中的大部分也出现在其它头文件中。

库变量

下面是头文件 stddef.h 中定义的变量类型：

变量	描述
ptrdiff_t	这是有符号整数类型，它是两个指针相减的结果。
size_t	这是无符号整数类型，它是 **sizeof** 关键字的结果。
wchar_t	这是一个宽字符常量大小的整数类型。

库宏

下面是头文件 stddef.h 中定义的宏：

宏	描述
NULL	这个宏是一个空指针常量的值。
offsetof(type, member-designator)	这会生成一个类型为 size*t* 的整型常量，它是一个结构成员相对于结构开头的字节偏移量。成员是由 *_member-designator* 给定的，结构的名称是在 *type* 中给定的。

C 库宏 - NULL

描述

C 库宏 **NULL** 是一个空指针常量的值。它可以被定义为 **((void*)0), 0** 或 **0L**，这取决于编译器供应商。

声明

下面是取决于编译器的 NULL 宏的声明。

```
#define NULL ((char *)0)

或

#define NULL 0L

或

#define NULL 0
```

参数

- **NA**

返回值

- **NA**

实例

下面的实例演示了 NULL 宏的用法。

```
#include <stddef.h>
#include <stdio.h>

int main ()
{
   FILE *fp;

   fp = fopen("file.txt", "r");
   if( fp != NULL )
   {
      printf("成功打开文件 file.txt\n");
      fclose(fp);
   }

   fp = fopen("nofile.txt", "r");
   if( fp == NULL )
   {
      printf("不能打开文件 nofile.txt\n");
   }

   return(0);
}
```

假设文件 **file.txt** 已存在，但是 **nofile.txt** 不存在。让我们编译并运行上面的程序，这将产生以下结果：

```
成功打开文件 file.txt
不能打开文件 nofile.txt
```

C 库宏 - offsetof()

描述

C 库宏 **offsetof(type, member-designator)** 会生成一个类型为 **size_t** 的整型常量，它是一个结构成员相对于结构开头的字节偏移量。成员是由 member-designator 给定的，结构的名称是在 type 中给定的。

声明

下面是 offsetof() 宏的声明。

```
offsetof(type, member-designator)
```

参数

- **type** -- 这是一个 class 类型，其中，member-designator 是一个有效的成员指示器。
- **member-designator** -- 这是一个 class 类型的成员指示器。

返回值

该宏返回类型为 **size_t** 的值，表示 type 中成员的偏移量。

实例

下面的实例演示了 offsetof() 宏的用法。

```
#include <stddef.h>
#include <stdio.h>

struct address {
   char name[50];
   char street[50];
   int phone;
};

int main()
{
   printf("address 结构中的 name 偏移 = %d 字节。\n",
   offsetof(struct address, name));

   printf("address 结构中的 street 偏移 = %d 字节。\n",
   offsetof(struct address, street));

   printf("address 结构中的 phone 偏移 = %d 字节。\n",
   offsetof(struct address, phone));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
address 结构中的 name 偏移 = 0 字节。
address 结构中的 street 偏移 = 50 字节。
address 结构中的 phone 偏移 = 100 字节。
```

C 标准库 - <stdio.h>

简介

stdio .h 头文件定义了三个变量类型、一些宏和各种函数来执行输入和输出。

库变量

下面是头文件 stdio.h 中定义的变量类型：

变量	描述
size_t	这是无符号整数类型，它是 **sizeof** 关键字的结果。
FILE	这是一个适合存储文件流信息的对象类型。
fpos_t	这是一个适合存储文件中任何位置的对象类型。

库宏

下面是头文件 stdio.h 中定义的宏：

宏	描述
NULL	这个宏是一个空指针常量的值。
_IOFBF、_IOLBF 和 _IONBF	这些宏扩展了带有特定值的整型常量表达式，并适用于 **setvbuf** 函数的第三个参数。
BUFSIZ	这个宏是一个整数，该整数代表了 **setbuf** 函数使用的缓冲区大小。
EOFM	这个宏是一个表示已经到达文件结束的负整数。
FOPEN_MAX	这个宏是一个整数，该整数代表了系统可以同时打开的文件数量。
FILENAME_MAX	这个宏是一个整数，该整数代表了字符数组可以存储的文件名的最大长度。如果实现没有任何限制，则该值应为推荐的最大值。
L_tmpnam	这个宏是一个整数，该整数代表了字符数组可以存储的由 tmpnam 函数创建的临时文件名的最大长度。
SEEK_CUR、SEEK_END 和 SEEK_SET	这些宏是在These macros are used in the **fseek** 函数中使用，用于在一个文件中定位不同的位置。
TMP_MAX	这个宏是 tmpnam 函数可生成的独特文件名的最大数量。
stderr、stdin 和 stdout	这些宏是指向 FILE 类型的指针，分别对应于标准错误、标准输入和标准输出流。

库函数

下面是头文件 stdio.h 中定义的函数：

> 为了更好地理解函数，请按照下面的序列学习这些函数，因为第一个函数中创建的文件会在后续的函数中使用到。

函数	描述
int fclose(FILE *stream)	关闭流 stream。刷新所有的缓冲区。
void clearerr(FILE *stream)	清除给定流 stream 的文件结束和错误标识符。
int feof(FILE *stream)	测试给定流 stream 的文件结束标识符。
int ferror(FILE *stream)	测试给定流 stream 的错误标识符。
int fflush(FILE *stream)	刷新流 stream 的输出缓冲区。
int fgetpos(FILE *stream, fpos_t *pos)	获取流 stream 的当前文件位置，并把它写入到 pos。
FILE *fopen(const char	

*filename, const char *mode)	使用给定的模式 mode 打开 filename 所指向的文件。
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)	从给定流 stream 读取数据到 ptr 所指向的数组中。
FILE *freopen(const char *filename, const char *mode, FILE *stream)	把一个新的文件名 filename 与给定的打开的流 stream 关联，同时关闭流中的旧文件。
int fseek(FILE *stream, long int offset, int whence)	设置流 stream 的文件位置为给定的偏移 offset，参数 *offset* 意味着从给定的 *whence* 位置查找的字节数。
int fsetpos(FILE *stream, const fpos_t *pos)	设置给定流 stream 的文件位置为给定的位置。参数 *pos* 是由函数 fgetpos 给定的位置。
long int ftell(FILE *stream)	返回给定流 stream 的当前文件位置。
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)	把 ptr 所指向的数组中的数据写入到给定流 stream 中。
int remove(const char *filename)	删除给定的文件名 filename，以便它不再被访问。
int rename(const char *old_filename, const char *new_filename)	把 old_filename 所指向的文件名改为 new_filename。
void rewind(FILE *stream)	设置文件位置为给定流 stream 的文件的开头。
void setbuf(FILE *stream, char *buffer)	定义流 stream 应如何缓冲。
int setvbuf(FILE *stream, char *buffer, int mode, size_t size)	另一个定义流 stream 应如何缓冲的函数。
FILE *tmpfile(void)	以二进制更新模式(wb+)创建临时文件。
char *tmpnam(char *str)	生成并返回一个有效的临时文件名，该文件名之前是不存在的。
int fprintf(FILE *stream, const char *format, ...)	发送格式化输出到流 stream 中。
int printf(const char *format, ...)	发送格式化输出到标准输出 stdout。
int sprintf(char *str, const char *format, ...)	发送格式化输出到字符串。

int vfprintf(FILE *stream, const char *format, va_list arg)	使用参数列表发送格式化输出到流 stream 中。
int vprintf(const char *format, va_list arg)	使用参数列表发送格式化输出到标准输出 stdout。
int vsprintf(char *str, const char *format, va_list arg)	使用参数列表发送格式化输出到字符串。
int fscanf(FILE *stream, const char *format, ...)	从流 stream 读取格式化输入。
int scanf(const char *format, ...)	从标准输入 stdin 读取格式化输入。
int sscanf(const char *str, const char *format, ...)	从字符串读取格式化输入。
int fgetc(FILE *stream)	从指定的流 stream 获取下一个字符（一个无符号字符），并把位置标识符往前移动。
char *fgets(char *str, int n, FILE *stream)	从指定的流 stream 读取一行，并把它存储在 str 所指向的字符串内。当读取 **(n-1)** 个字符时，或者读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。
int fputc(int char, FILE *stream)	把参数 char 指定的字符（一个无符号字符）写入到指定的流 stream 中，并把位置标识符往前移动。
int fputs(const char *str, FILE *stream)	把字符串写入到指定的流 stream 中，但不包括空字符。
int getc(FILE *stream)	从指定的流 stream 获取下一个字符（一个无符号字符），并把位置标识符往前移动。
int getchar(void)	从标准输入 stdin 获取一个字符（一个无符号字符）。
char *gets(char *str)	从标准输入 stdin 读取一行，并把它存储在 str 所指向的字符串中。当读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。
int putc(int char, FILE *stream)	把参数 char 指定的字符（一个无符号字符）写入到指定的流 stream 中，并把位置标识符往前移动。
int putchar(int char)	把参数 char 指定的字符（一个无符号字符）写入到标准输出 stdout 中。
int puts(const char *str)	把一个字符串写入到标准输出 stdout，直到空字符，但不包括空字符。换行符会被追加到输出中。
int ungetc(int char, FILE *stream)	把字符 char（一个无符号字符）推入到指定的流 stream 中，以便它是下一个被读取到的字符。
void perror(const char *str)	把一个描述性错误消息输出到标准错误 stderr。首先输出字符串 str，后跟一个冒号，然后是一个空格。

C 库函数 - fclose()

描述

C 库函数 **int fclose(FILE *stream)** 关闭流 stream。刷新所有的缓冲区。

声明

下面是 fclose() 函数的声明。

```
int fclose(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象指定了要被关闭的流。

返回值

如果流成功关闭，则该方法返回零。如果失败，则返回 EOF。

实例

下面的实例演示了 fclose() 函数的用法。

```
#include <stdio.h>

int main()
{
   FILE *fp;

   fp = fopen("file.txt", "w");

   fprintf(fp, "%s", "这里是 w3cschool.cc");
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建一个文件 **file.txt**，然后写入下面的文本行，最后使用 **fclose()** 函数关闭文件。

```
这里是 w3cschool.cc
```

C 库函数 - clearerr()

描述

C 库函数 **void clearerr(FILE *stream)** 清除给定流 stream 的文件结束和错误标识符。

声明

下面是 clearerr() 函数的声明。

```
void clearerr(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。

返回值

这不会失败，且不会设置外部变量 errno，但是如果它检测到它的参数不是一个有效的流，则返回 -1，并设置 errno 为 EBADF。

实例

下面的实例演示了 clearerr() 函数的用法。

```
#include <stdio.h>

int main()
{
   FILE *fp;
   char c;

   fp = fopen("file.txt", "w");

   c = fgetc(fp);
   if( ferror(fp) )
   {
      printf("读取文件：file.txt 时发生错误\n");
   }
   clearerr(fp);
   if( ferror(fp) )
   {
      printf("读取文件：file.txt 时发生错误\n");
   }
   fclose(fp);

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它是一个空文件。让我们编译并运行上面的程序，因为我们试图读取一个以只写模式打开的文件，这将产生以下结果。

```
读取文件：file.txt 时发生错误
```

C 库函数 - feof()

描述

C 库函数 **int feof(FILE *stream)** 测试给定流 stream 的文件结束标识符。

声明

下面是 feof() 函数的声明。

```
int feof(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。

返回值

当设置了与流关联的文件结束标识符时，该函数返回一个非零值，否则返回零。

实例

下面的实例演示了 feof() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   if(fp == NULL)
   {
      perror("打开文件时发生错误");
      return(-1);
   }
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下所示。该文件将作为我们实例程序中的一个输入使用：

```
这里是 w3cschool.cc
```

让我们编译并运行上面的程序，这将产生以下结果：

```
这里是 w3cschool.cc
```

C 库函数 - ferror()

描述

C 库函数 **int ferror(FILE *stream)** 测试给定流 stream 的错误标识符。

声明

下面是 ferror() 函数的声明。

```
int ferror(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。

返回值

如果设置了与流关联的错误标识符，该函数返回一个非零值，否则返回一个零值。

实例

下面的实例演示了 ferror() 函数的用法。

```
#include <stdio.h>

int main()
{
   FILE *fp;
   char c;

   fp = fopen("file.txt", "w");

   c = fgetc(fp);
   if( ferror(fp) )
   {
      printf("读取文件：file.txt 时发生错误\n");
   }
   clearerr(fp);
   if( ferror(fp) )
   {
      printf("读取文件：file.txt 时发生错误\n");
   }
   fclose(fp);

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它是一个空文件。让我们编译并运行上面的程序，因为我们试图读取一个以只写模式打开的文件，这将产生以下结果。

```
读取文件：file.txt 时发生错误
```

C 库函数 - fflush()

描述

C 库函数 **int fflush(FILE *stream)** 刷新流 stream 的输出缓冲区。

声明

下面是 fflush() 函数的声明。

```
int fflush(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象指定了一个缓冲流。

返回值

如果成功，该函数返回零值。如果发生错误，则返回 EOF，且设置错误标识符（即 feof）。

实例

下面的实例演示了 fflush() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main()
{

   char buff[1024];

   memset( buff, '\0', sizeof( buff ));

   fprintf(stdout, "启用全缓冲\n");
   setvbuf(stdout, buff, _IOFBF, 1024);

   fprintf(stdout, "这里是 w3cschool.cc\n");
   fprintf(stdout, "该输出将保存到 buff\n");
   fflush( stdout );

   fprintf(stdout, "这将在编程时出现\n");
   fprintf(stdout, "最后休眠五秒钟\n");

   sleep(5);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果。在这里，程序把缓冲输出保存到 **buff**，直到首次调用 **fflush()** 为止，然后开始缓冲输出，最后休眠 5 秒钟。它会在程序结束之前，发送剩余的输出到 STDOUT。

```
启用全缓冲
这里是 w3cschool.cc
该输出将保存到 buff
这将在编程时出现
最后休眠五秒钟
```

C 库函数 - fgetpos()

描述

C 库函数 **int fgetpos(FILE *stream, fpos_t *pos)** 获取流 **stream** 的当前文件位置，并把它写入到 **pos**。

声明

下面是 fgetpos() 函数的声明。

```
int fgetpos(FILE *stream, fpos_t *pos)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
- **pos** -- 这是指向 fpos_t 对象的指针。

返回值

如果成功，该函数返回零。如果发生错误，则返回非零值。

实例

下面的实例演示了 fgetpos() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   fpos_t position;

   fp = fopen("file.txt","w+");
   fgetpos(fp, &position);
   fputs("Hello, World!", fp);

   fsetpos(fp, &position);
   fputs("这将覆盖之前的内容", fp);
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建一个文件 **file.txt**，它的内容如下。首先我们使用 **fgetpos()** 函数获取文件的初始位置，接着我们向文件写入 *Hello, World!*，然后我们使用 **fsetpos()** 函数来重置写指针到文件的开头，重写文件为下列内容：

```
这将覆盖之前的内容
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;
   int n = 0;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }

   fclose(fp);

   return(0);
}
```

C 库函数 - fopen()

描述

C 库函数 **FILE *fopen(const char* filename, const char *mode)** 使用给定的模式 **mode** 打开 **filename** 所指向的文件。

声明

下面是 fopen() 函数的声明。

```
FILE *fopen(const char *filename, const char *mode)
```

参数

- **filename** -- 这是 C 字符串，包含了要打开的文件名称。
- **mode** -- 这是 C 字符串，包含了文件访问模式，模式如下：

模式	描述
"r"	打开一个用于读取的文件。该文件必须存在。
"w"	创建一个用于写入的空文件。如果文件名称与已存在的文件相同，则会删除已有文件的内容，文件被视为一个新的空文件。
"a"	追加到一个文件。写操作向文件末尾追加数据。如果文件不存在，则创建文件。
"r+"	打开一个用于更新的文件，可读取也可写入。该文件必须存在。
"w+"	创建一个用于读写的空文件。
"a+"	打开一个用于读取和追加的文件。

返回值

该函数返回一个 FILE 指针。否则返回 NULL，且设置全局变量 errno 来标识错误。

实例

下面的实例演示了 fopen() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fprintf(fp, "%s %s %s %d", "We", "are", "in", 2014);

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建一个带有一下内容的文件 **file.txt**：

```
We are in 2014
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - fread()

描述

C 库函数 **size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)** 从给定流 **stream** 读取数据到 **ptr** 所指向的数组中。

声明

下面是 fread() 函数的声明。

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

参数

- **ptr** -- 这是指向带有最小尺寸 *size*nmemb* 字节的内存块的指针。
- **size** -- 这是要读取的每个元素的大小，以字节为单位。
- **nmemb** -- 这是元素的个数，每个元素的大小为 size 字节。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象指定了一个输入流。

返回值

成功读取的元素总数会以 size_t 对象返回，size_t 对象是一个整型数据类型。如果总数与 nmemb 参数不同，则可能发生了一个错误或者到达了文件末尾。

实例

下面的实例演示了 fread() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main()
{
   FILE *fp;
   char c[] = "This is w3cschool";
   char buffer[20];

   /* 打开文件用于读写 */
   fp = fopen("file.txt", "w+");

   /* 写入数据到文件 */
   fwrite(c, strlen(c) + 1, 1, fp);

   /* 查找文件的开头 */
   fseek(fp, SEEK_SET, 0);

   /* 读取并显示数据 */
   fread(buffer, strlen(c)+1, 1, fp);
   printf("%s\n", buffer);
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建一个文件 **file.txt**，然后写入内容 *this is w3cschool*。接下来我们使用 **fseek()** 函数来重置写指针到文件的开头，文件内容如下所示：

```
This is w3cschool
```

C 库函数 - freopen()

描述

C 库函数 **FILE *freopen(const char* filename, const char *mode, FILE* stream)** 把一个新的文件名 **filename** 与给定的打开的流 **stream** 关联，同时关闭流中的旧文件。

声明

下面是 freopen() 函数的声明。

```
FILE *freopen(const char *filename, const char *mode, FILE *stream)
```

参数

- **filename** -- 这是 C 字符串，包含了要打开的文件名称。
- **mode** -- 这是 C 字符串，包含了文件访问模式，模式如下：

模式	描述
"r"	打开一个用于读取的文件。该文件必须存在。
"w"	创建一个用于写入的空文件。如果文件名称与已存在的文件相同，则会删除已有文件的内容，文件被视为一个新的空文件。
"a"	追加到一个文件。写操作向文件末尾追加数据。如果文件不存在，则创建文件。
"r+"	打开一个用于更新的文件，可读取也可写入。该文件必须存在。
"w+"	创建一个用于读写的空文件。
"a+"	打开一个用于读取和追加的文件。

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要被重新打开的流。

返回值

如果文件成功打开，则函数返回一个指针，指向用于标识流的对象。否则，返回空指针。

实例

下面的实例演示了 freopen() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   printf("该文本重定向到 stdout\n");

   fp = freopen("file.txt", "w+", stdout);

   printf("该文本重定向到 file.txt\n");

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将发送下列行到标准输出 STDOUT，因为起初我们并没有打开标准输出：

```
该文本重定向到 stdout
```

在调用 **freopen()** 之后，它会关联标准输出 STDOUT 到文件 **file.txt**，无论我们在标准输出 STDOUT 中写了什么都会被写入 file.txt，所以文件 file.txt 将有以下内容。

```
该文本重定向到 file.txt
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - fseek()

描述

C 库函数 **int fseek(FILE *stream, long int offset, int whence)** 设置流 **stream** 的文件位置为给定的偏移 **offset**，参数 offset 意味着从给定的 **whence** 位置查找的字节数。

声明

下面是 fseek() 函数的声明。

```
int fseek(FILE *stream, long int offset, int whence)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
- **offset** -- 这是相对 whence 的偏移量，以字节为单位。
- **whence** -- 这是表示开始添加偏移 offset 的位置。它一般指定为下列常量之一：

常量	描述
SEEK_SET	文件的开头
SEEK_CUR	文件指针的当前位置
SEEK_END	文件的末尾

返回值

如果成功，则该函数返回零，否则返回非零值。

实例

下面的实例演示了 fseek() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   fp = fopen("file.txt","w+");
   fputs("This is w3cschool.cc", fp);

   fseek( fp, 7, SEEK_SET );
   fputs(" C Programming Langauge", fp);
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建文件 **file.txt**，它的内容如下。最初程序创建文件和写入 *This is w3cschool.cc*，但是之后我们在第七个位置重置了写指针，并使用 puts() 语句来重写文件，内容如下：

```
This is C Programming Langauge
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - fsetpos()

描述

C 库函数 **int fsetpos(FILE *stream, const fpos_t* pos)** 设置给定流 **stream** 的文件位置为给定的位置。参数 **pos** 是由函数 fgetpos 给定的位置。

声明

下面是 fsetpos() 函数的声明。

```
int fsetpos(FILE *stream, const fpos_t *pos)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
- **pos** -- 这是指向 fpos_t 对象的指针，该对象包含了之前通过 fgetpos 获得的位置。

返回值

如果成功，该函数返回零值，否则返回非零值，并设置全局变量 **errno** 为一个正值，该值可通过 perror 来解释。

实例

下面的实例演示了 fsetpos() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   fpos_t position;

   fp = fopen("file.txt","w+");
   fgetpos(fp, &position);
   fputs("Hello, World!", fp);

   fsetpos(fp, &position);
   fputs("这将覆盖之前的内容", fp);
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建一个文件 **file.txt**，它的内容如下。首先我们使用 **fgetpos()** 函数获取文件的初始位置，接着我们向文件写入 *Hello, World!*，然后我们使用 **fsetpos()** 函数来重置写指针到文件的开头，重写文件为下列内容：

```
这将覆盖之前的内容
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - ftell()

描述

C 库函数 **long int ftell(FILE *stream)** 返回给定流 stream 的当前文件位置。

声明

下面是 ftell() 函数的声明。

```
long int ftell(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。

返回值

该函数返回位置标识符的当前值。如果发生错误，则返回 -1L，全局变量 errno 被设置为一个正值。

实例

下面的实例演示了 ftell() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int len;

   fp = fopen("file.txt", "r");
   if( fp == NULL )
   {
      perror ("打开文件错误");
      return(-1);
   }
   fseek(fp, 0, SEEK_END);

   len = ftell(fp);
   fclose(fp);

   printf("file.txt 的总大小 = %d 字节\n", len);

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下：

```
This is w3cschool.cc
```

让我们编译并运行上面的程序，如果文件内容如上所示，这将产生以下结果，否则会根据文件内容给出不同的结果：

```
file.txt 的总大小 = 21 字节
```

C 库函数 - fwrite()

描述

C 库函数 **size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE* stream)** 把 **ptr** 所指向的数组中的数据写入到给定流 **stream** 中。

声明

下面是 fwrite() 函数的声明。

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

参数

- **ptr** -- 这是指向要被写入的元素数组的指针。
- **size** -- 这是要被写入的每个元素的大小，以字节为单位。
- **nmemb** -- 这是元素的个数，每个元素的大小为 size 字节。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象指定了一个输出流。

返回值

如果成功，该函数返回一个 size_t 对象，表示元素的总数，该对象时一个整型数据类型。如果该数字与 nmemb 参数不同，则会显示一个错误。

实例

下面的实例演示了 fwrite() 函数的用法。

```
#include<stdio.h>

int main ()
{
   FILE *fp;
   char str[] = "This is w3cschool.cc";

   fp = fopen( "file.txt" , "w" );
   fwrite(str , 1 , sizeof(str) , fp );

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建一个文件 **file.txt**，它的内容如下：

```
This is w3cschool.cc
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - remove()

描述

C 库函数 **int remove(const char *filename)** 删除给定的文件名 **filename**，以便它不再被访问。

声明

下面是 remove() 函数的声明。

```
int remove(const char *filename)
```

参数

- **filename** -- 这是 C 字符串，包含了要被删除的文件名称。

返回值

如果成功，则返回零。如果错误，则返回 -1，并设置 errno。

实例

下面的实例演示了 remove() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   int ret;
   FILE *fp;
   char filename[] = "file.txt";

   fp = fopen(filename, "w");

   fprintf(fp, "%s", "这里是 w3cschool.cc");
   fclose(fp);

   ret = remove(filename);

   if(ret == 0)
   {
      printf("文件删除成功");
   }
   else
   {
      printf("错误：不能删除该文件");
   }

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下。我们将使用上面的程序来删除该文件。让我们编译并运行上面的程序，这将生成下面的消息，且文件被永久删除。

```
文件删除成功
```

C 库函数 - rename()

描述

C 库函数 **int rename(const char *old_filename, const char* new_filename)** 把 **old_filename** 所指向的文件名改为 **new_filename**。

声明

下面是 rename() 函数的声明。

```
int rename(const char *old_filename, const char *new_filename)
```

参数

- **old_filename** -- 这是 C 字符串，包含了要被重命名/移动的文件名称。
- **new_filename** -- 这是 C 字符串，包含了文件的新名称。

返回值

如果成功，则返回零。如果错误，则返回 -1，并设置 errno。

实例

下面的实例演示了 rename() 函数的用法。

```
#include <stdio.h>

int main ()
{
   int ret;
   char oldname[] = "file.txt";
   char newname[] = "newfile.txt";

   ret = rename(oldname, newname);

   if(ret == 0)
   {
      printf("文件重命名成功");
   }
   else
   {
      printf("错误：不能重命名该文件");
   }

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下。我们将使用上面的程序来重命名该文件。让我们编译并运行上面的程序，这将生成下面的消息，且文件被重命名为 **newfile.txt** 文件。

```
文件重命名成功
```

C 库函数 - rewind()

描述

C 库函数 **void rewind(FILE *stream)** 设置文件位置为给定流 **stream** 的文件的开头。

声明

下面是 rewind() 函数的声明。

```
void rewind(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。

返回值

该函数不返回任何值。

实例

下面的实例演示了 rewind() 函数的用法。

```
#include <stdio.h>

int main()
{
   char str[] = "This is w3cschool.cc";
   FILE *fp;
   int ch;

   /* 首先让我们在文件中写入一些内容 */
   fp = fopen( "file.txt" , "w" );
   fwrite(str , 1 , sizeof(str) , fp );
   fclose(fp);

   fp = fopen( "file.txt" , "r" );
   while(1)
   {
      ch = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", ch);
   }
   rewind(fp);
   printf("\n");
   while(1)
   {
      ch = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", ch);

   }
   fclose(fp);

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下：

```
This is w3cschool.cc
```

让我们编译并运行上面的程序，这将产生以下结果：

```
This is w3cschool.cc
This is w3cschool.cc
```

C 库函数 - setbuf()

描述

C 库函数 **void setbuf(FILE *stream, char* buffer)** 定义流 stream 应如何缓冲。该函数应在与流 stream 相关的文件被打开时，且还未发生任何输入或输出操作之前被调用一次。

声明

下面是 setbuf() 函数的声明。

```
void setbuf(FILE *stream, char *buffer)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了一个打开的流。
- **buffer** -- 这是分配给用户的缓冲，它的长度至少为 BUFSIZ 字节，BUFSIZ 是一个宏常量，表示数组的长度。

返回值

该函数不返回任何值。

实例

下面的实例演示了 setbuf() 函数的用法。

```
#include <stdio.h>

int main()
{
   char buf[BUFSIZ];

   setbuf(stdout, buf);
   puts("This is w3cschool");

   fflush(stdout);
   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果。在这里，程序在即将输出的时候，发送输出到 STDOUT，否则它将缓冲输出。您也可以使用 fflush() 函数来 to 刷新输出。

```
This is w3cschool
```

C 库函数 - tmpfile()

描述

C 库函数 **FILE *tmpfile(void)** 以二进制更新模式(wb+)创建临时文件。被创建的临时文件会在流关闭的时候或者在程序终止的时候自动删除。

声明

下面是 tmpfile() 函数的声明。

```
FILE *tmpfile(void)
```

参数

- **NA**

返回值

如果成功，该函数返回一个指向被创建的临时文件的流指针。如果文件未被创建，则返回 NULL。

实例

下面的实例演示了 tmpfile() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   fp = tmpfile();
   printf("临时文件被创建\n");

   /* 您可以在这里使用临时文件 */

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，它将在 /tmp 文件夹中创建一个临时文件，但是一旦程序退出，临时文件会被自动删除，且程序会产生以下结果：

```
临时文件被创建
```

C 库函数 - tmpnam()

描述

C 库函数 **char *tmpnam(char *str)** 生成并返回一个有效的临时文件名，该文件名之前是不存在的。如果 **str** 为空，则只会返回临时文件名。

声明

下面是 tmpnam() 函数的声明。

```
char *tmpnam(char *str)
```

参数

- **str** -- 这是一个指向字符数组的指针，其中，临时文件名将被存储为 C 字符串。

返回值

- 一个指向 C 字符串的指针，该字符串存储了临时文件名。如果 str 是一个空指针，则该指针指向一个内部缓冲区，缓冲区在下一次调用函数时被覆盖。
- 如果 str 不是一个空指针，则返回 str。如果函数未能成功创建可用的文件名，则返回一个空指针。

实例

下面的实例演示了 tmpnam() 函数的用法。

```
#include <stdio.h>

int main()
{
   char buffer[L_tmpnam];
   char *ptr;

   tmpnam(buffer);
   printf("临时名称 1: %s\n", buffer);

   ptr = tmpnam(NULL);
   printf("临时名称 2: %s\n", ptr);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
临时名称 1: /tmp/filebaalTb
临时名称 2: /tmp/filedCIbb0
```

C 库函数 - fprintf()

描述

C 库函数 **int fprintf(FILE *stream, const char *format, ...)** 发送格式化输出到流 stream 中。

声明

下面是 fprintf() 函数的声明。

```
int fprintf(FILE *stream, const char *format, ...)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
- **format** -- 这是 C 字符串，包含了要被写入到流 stream 中的文本。它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。format 标签属性是 **%[flags][width][.precision][length]specifier**，具体讲解如下：

specifier（说明符）	输出
c	字符
d 或 i	有符号十进制整数
e	使用 e 字符的科学科学记数法（尾数和指数）
E	使用 E 字符的科学科学记数法（尾数和指数）
f	十进制浮点数
g	自动选择 %e 或 %f 中合适的表示法
G	自动选择 %E 或 %f 中合适的表示法
o	有符号八进制
s	字符的字符串
u	无符号十进制整数
x	无符号十六进制整数
X	无符号十六进制整数（大写字母）
p	指针地址
n	无输出
%	字符

flags（标识）	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
(space)	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width（宽度）	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision（精度）	描述
.number	对于整数说明符（d、i、o、u、x、X）：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。 对于 e、E 和 f 说明符：要在小数点后输出的小数位数。 对于 g 和 G 说明符：要输出的最大有效位数。 对于 s: 要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。 对于 c 类型：没有任何影响。 当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。
.*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

length（长度）	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）。
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）。
L	参数被解释为长双精度型（仅适用于浮点数说明符：e、E、f、g 和 G）。

- 附加参数 -- 根据不同的 format 字符串，函数可能需要一系列的附加参数，每个参数包含了一个要被插入的值，替换了 format 参数中指定的每个 % 标签。参数的个数应与 % 标签的个数相同。

返回值

如果成功，则返回写入的字符总数，否则返回一个负数。

实例

下面的实例演示了 fprintf() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fprintf(fp, "%s %s %s %d", "We", "are", "in", 2014);

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建文件 **file.txt**，它的内容如下：

```
We are in 2014
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - printf()

描述

C 库函数 **int printf(const char *format, ...)** 发送格式化输出到标准输出 stdout。

声明

下面是 printf() 函数的声明。

```
int printf(const char *format, ...)
```

参数

- **format** -- 这是字符串，包含了要被写入到标准输出 stdout 的文本。它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。format 标签属性是 **%[flags][width][.precision][length]specifier**，具体讲解如下：

specifier（说明符）	输出
c	字符
d 或 i	有符号十进制整数
e	使用 e 字符的科学科学记数法（尾数和指数）
E	使用 E 字符的科学科学记数法（尾数和指数）
f	十进制浮点数
g	自动选择 %e 或 %f 中合适的表示法
G	自动选择 %E 或 %f 中合适的表示法
o	有符号八进制
s	字符的字符串
u	无符号十进制整数
x	无符号十六进制整数
X	无符号十六进制整数（大写字母）
p	指针地址
n	无输出
%	字符

flags（标识）	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
(space)	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width（宽度）	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision（精度）	描述
.number	对于整数说明符（d、i、o、u、x、X）：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。 对于 e、E 和 f 说明符：要在小数点后输出的小数位数。 对于 g 和 G 说明符：要输出的最大有效位数。 对于 s: 要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。 对于 c 类型：没有任何影响。 当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。
.*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

length（长度）	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）。
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）。
L	参数被解释为长双精度型（仅适用于浮点数说明符：e、E、f、g 和 G）。

- 附加参数 -- 根据不同的 format 字符串，函数可能需要一系列的附加参数，每个参数包含了一个要被插入的值，替换了 format 参数中指定的每个 % 标签。参数的个数应与 % 标签的个数相同。

返回值

如果成功，则返回写入的字符总数，否则返回一个负数。

实例

下面的实例演示了 printf() 函数的用法。

```
#include <stdio.h>

int main ()
{
   int ch;

   for( ch = 75 ; ch <= 100; ch++ ) {
      printf("ASCII 值 = %d, 字符 = %c\n", ch , ch );
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
ASCII 值 = 75, 字符 = K
ASCII 值 = 76, 字符 = L
ASCII 值 = 77, 字符 = M
ASCII 值 = 78, 字符 = N
ASCII 值 = 79, 字符 = O
ASCII 值 = 80, 字符 = P
ASCII 值 = 81, 字符 = Q
ASCII 值 = 82, 字符 = R
ASCII 值 = 83, 字符 = S
ASCII 值 = 84, 字符 = T
ASCII 值 = 85, 字符 = U
ASCII 值 = 86, 字符 = V
ASCII 值 = 87, 字符 = W
ASCII 值 = 88, 字符 = X
ASCII 值 = 89, 字符 = Y
ASCII 值 = 90, 字符 = Z
ASCII 值 = 91, 字符 = [
ASCII 值 = 92, 字符 = \
ASCII 值 = 93, 字符 = ]
ASCII 值 = 94, 字符 = ^
ASCII 值 = 95, 字符 = _
ASCII 值 = 96, 字符 = `
ASCII 值 = 97, 字符 = a
ASCII 值 = 98, 字符 = b
ASCII 值 = 99, 字符 = c
ASCII 值 = 100, 字符 = d
```

C 库函数 - sprintf()

描述

C 库函数 **int sprintf(char *str, const char* format, ...)** 发送格式化输出到 **str** 所指向的字符串。

声明

下面是 sprintf() 函数的声明。

```
int sprintf(char *str, const char *format, ...)
```

参数

- **str** -- 这是指向一个字符数组的指针，该数组存储了 C 字符串。
- **format** -- 这是字符串，包含了要被写入到字符串 str 的文本。它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。format 标签属性是 **%[flags][width][.precision][length]specifier**，具体讲解如下：

specifier（说明符）	输出
c	字符
d 或 i	有符号十进制整数
e	使用 e 字符的科学科学记数法（尾数和指数）
E	使用 E 字符的科学科学记数法（尾数和指数）
f	十进制浮点数
g	自动选择 %e 或 %f 中合适的表示法
G	自动选择 %E 或 %f 中合适的表示法
o	有符号八进制
s	字符的字符串
u	无符号十进制整数
x	无符号十六进制整数
X	无符号十六进制整数（大写字母）
p	指针地址
n	无输出
%	字符

flags（标识）	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
(space)	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width（宽度）	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision（精度）	描述
.number	对于整数说明符（d、i、o、u、x、X）：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。 对于 e、E 和 f 说明符：要在小数点后输出的小数位数。 对于 g 和 G 说明符：要输出的最大有效位数。 对于 s: 要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。 对于 c 类型：没有任何影响。 当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。
.*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

length（长度）	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）。
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）。
L	参数被解释为长双精度型（仅适用于浮点数说明符：e、E、f、g 和 G）。

- 附加参数 -- 根据不同的 format 字符串，函数可能需要一系列的附加参数，每个参数包含了一个要被插入的值，替换了 format 参数中指定的每个 % 标签。参数的个数应与 % 标签的个数相同。

返回值

如果成功，则返回写入的字符总数，不包括字符串追加在字符串末尾的空字符。如果失败，则返回一个负数。

实例

下面的实例演示了 sprintf() 函数的用法。

```
#include <stdio.h>
#include <math.h>

int main()
{
   char str[80];

   sprintf(str, "Pi 的值 = %f", M_PI);
   puts(str);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Pi 的值 = 3.141593
```

C 库函数 - vfprintf()

描述

C 库函数 **int vfprintf(FILE *stream, const char *format, va_list arg)** 使用参数列表发送格式化输出到流 stream 中。

声明

下面是 vfprintf() 函数的声明。

```
int vfprintf(FILE *stream, const char *format, va_list arg)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
- **format** -- 这是 C 字符串，包含了要被写入到流 stream 中的文本。它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。format 标签属性是 **%[flags][width][.precision][length]specifier**，具体讲解如下：

specifier（说明符）	输出
c	字符
d 或 i	有符号十进制整数
e	使用 e 字符的科学科学记数法（尾数和指数）
E	使用 E 字符的科学科学记数法（尾数和指数）
f	十进制浮点数
g	自动选择 %e 或 %f 中合适的表示法
G	自动选择 %E 或 %f 中合适的表示法
o	有符号八进制
s	字符的字符串
u	无符号十进制整数
x	无符号十六进制整数
X	无符号十六进制整数（大写字母）
p	指针地址
n	无输出
%	字符

flags（标识）	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
(space)	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width（宽度）	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision（精度）	描述
.number	对于整数说明符（d、i、o、u、x、X）：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。 对于 e、E 和 f 说明符：要在小数点后输出的小数位数。 对于 g 和 G 说明符：要输出的最大有效位数。 对于 s: 要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。 对于 c 类型：没有任何影响。 当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。
.*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

length（长度）	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）。
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）。
L	参数被解释为长双精度型（仅适用于浮点数说明符：e、E、f、g 和 G）。

- **arg** -- 一个表示可变参数列表的对象。这应被 <stdarg> 中定义的 va_start 宏初始化。

返回值

如果成功，则返回写入的字符总数，否则返回一个负数。

实例

下面的实例演示了 vfprintf() 函数的用法。

```
#include <stdio.h>
#include <stdarg.h>

void WriteFrmtd(FILE *stream, char *format, ...)
{
   va_list args;

   va_start(args, format);
   vfprintf(stream, format, args);
   va_end(args);
}

int main ()
{
   FILE *fp;

   fp = fopen("file.txt","w");

   WriteFrmtd(fp, "This is just one argument %d \n", 10);

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将打开当前目录中的文件 **file.txt**，并写入以下内容：

```
This is just one argument 10
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
         break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - vprintf()

描述

C 库函数 **int vprintf(const char *format, va_list arg)** 使用参数列表发送格式化输出到标准输出 stdout。

声明

下面是 vprintf() 函数的声明。

```
int vprintf(const char *format, va_list arg)
```

参数

- **format** -- 这是字符串，包含了要被写入到标准输出 stdout 的文本。它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。format 标签属性是 **%[flags][width][.precision][length]specifier**，具体讲解如下：

specifier（说明符）	输出
c	字符
d 或 i	有符号十进制整数
e	使用 e 字符的科学科学记数法（尾数和指数）
E	使用 E 字符的科学科学记数法（尾数和指数）
f	十进制浮点数
g	自动选择 %e 或 %f 中合适的表示法
G	自动选择 %E 或 %f 中合适的表示法
o	有符号八进制
s	字符的字符串
u	无符号十进制整数
x	无符号十六进制整数
X	无符号十六进制整数（大写字母）
p	指针地址
n	无输出
%	字符

flags（标识）	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
(space)	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width（宽度）	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision（精度）	描述
.number	对于整数说明符（d、i、o、u、x、X）：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。 对于 e、E 和 f 说明符：要在小数点后输出的小数位数。 对于 g 和 G 说明符：要输出的最大有效位数。 对于 s: 要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。 对于 c 类型：没有任何影响。 当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。
.*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

length（长度）	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）。
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）。
L	参数被解释为长双精度型（仅适用于浮点数说明符：e、E、f、g 和 G）。

- **arg** -- 一个表示可变参数列表的对象。这应被 <stdarg> 中定义的 va_start 宏初始化。

返回值

如果成功，则返回写入的字符总数，否则返回一个负数。

实例

下面的实例演示了 vprintf() 函数的用法。

```
#include <stdio.h>
#include <stdarg.h>

void WriteFrmtd(char *format, ...)
{
   va_list args;

   va_start(args, format);
   vprintf(format, args);
   va_end(args);
}

int main ()
{
   WriteFrmtd("%d variable argument\n", 1);
   WriteFrmtd("%d variable %s\n", 2, "arguments");

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
1 variable argument
2 variable arguments
```

C 库函数 - vsprintf()

描述

C 库函数 **int vsprintf(char *str, const char* format, va_list arg)** 使用参数列表发送格式化输出到字符串。

声明

下面是 vsprintf() 函数的声明。

```
int vsprintf(char *str, const char *format, va_list arg)
```

参数

- **str** -- 这是指向一个字符数组的指针，该数组存储了 C 字符串。
- **format** -- 这是字符串，包含了要被写入到字符串 str 的文本。它可以包含嵌入的 format 标签，format 标签可被随后的附加参数中指定的值替换，并按需求进行格式化。format 标签属性是 **%[flags][width][.precision][length]specifier**，具体讲解如下：

specifier（说明符）	输出
c	字符
d 或 i	有符号十进制整数
e	使用 e 字符的科学科学记数法（尾数和指数）
E	使用 E 字符的科学科学记数法（尾数和指数）
f	十进制浮点数
g	自动选择 %e 或 %f 中合适的表示法
G	自动选择 %E 或 %f 中合适的表示法
o	有符号八进制
s	字符的字符串
u	无符号十进制整数
x	无符号十六进制整数
X	无符号十六进制整数（大写字母）
p	指针地址
n	无输出
%	字符

flags（标识）	描述
-	在给定的字段宽度内左对齐，默认是右对齐（参见 width 子说明符）。
+	强制在结果之前显示加号或减号（+ 或 -），即正数前面会显示 + 号。默认情况下，只有负数前面会显示一个 - 号。
(space)	如果没有写入任何符号，则在该值前面插入一个空格。
#	与 o、x 或 X 说明符一起使用时，非零值前面会分别显示 0、0x 或 0X。 与 e、E 和 f 一起使用时，会强制输出包含一个小数点，即使后边没有数字时也会显示小数点。默认情况下，如果后边没有数字时候，不会显示显示小数点。 与 g 或 G 一起使用时，结果与使用 e 或 E 时相同，但是尾部的零不会被移除。
0	在指定填充 padding 的数字左边放置零（0），而不是空格（参见 width 子说明符）。

width（宽度）	描述
(number)	要输出的字符的最小数目。如果输出的值短于该数，结果会用空格填充。如果输出的值长于该数，结果不会被截断。
*	宽度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

.precision（精度）	描述
.number	对于整数说明符（d、i、o、u、x、X）：precision 指定了要写入的数字的最小位数。如果写入的值短于该数，结果会用前导零来填充。如果写入的值长于该数，结果不会被截断。精度为 0 意味着不写入任何字符。 对于 e、E 和 f 说明符：要在小数点后输出的小数位数。 对于 g 和 G 说明符：要输出的最大有效位数。 对于 s: 要输出的最大字符数。默认情况下，所有字符都会被输出，直到遇到末尾的空字符。 对于 c 类型：没有任何影响。 当未指定任何精度时，默认为 1。如果指定时不带有一个显式值，则假定为 0。
.*	精度在 format 字符串中未指定，但是会作为附加整数值参数放置于要被格式化的参数之前。

length（长度）	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）。
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）。
L	参数被解释为长双精度型（仅适用于浮点数说明符：e、E、f、g 和 G）。

- **arg** -- 一个表示可变参数列表的对象。这应被 <stdarg> 中定义的 va_start 宏初始化。

返回值

如果成功，则返回写入的字符总数，否则返回一个负数。

实例

下面的实例演示了 vsprintf() 函数的用法。

```
#include <stdio.h>
#include <stdarg.h>

char buffer[80];
int vspfunc(char *format, ...)
{
   va_list aptr;
   int ret;

   va_start(aptr, format);
   ret = vsprintf(buffer, format, aptr);
   va_end(aptr);

   return(ret);
}

int main()
{
   int i = 5;
   float f = 27.0;
   char str[50] = "w3cschool.cc";

   vspfunc("%d %f %s", i, f, str);
   printf("%s\n", buffer);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
5 27.000000 w3cschool.cc
```

C 库函数 - fscanf()

描述

C 库函数 **int fscanf(FILE *stream, const char *format, ...)** 从流 stream 读取格式化输入。

声明

下面是 fscanf() 函数的声明。

```
int fscanf(FILE *stream, const char *format, ...)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
- **format** -- 这是 C 字符串，包含了以下各项中的一个或多个：空格字符、非空格字符 和 *format* 说明符。
  format 说明符形式为 **[=%[*][width][modifiers]type=]**，具体讲解如下：

参数	描述
*	这是一个可选的星号，表示数据是从流 stream 中读取的，但是可以被忽视，即它不存储在对应的参数中。
width	这指定了在当前读取操作中读取的最大字符数。
modifiers	为对应的附加参数所指向的数据指定一个不同于整型（针对 d、i 和 n）、无符号整型（针对 o、u 和 x）或浮点型（针对 e、f 和 g）的大小： h ：短整型（针对 d、i 和 n），或无符号短整型（针对 o、u 和 x） l ：长整型（针对 d、i 和 n），或无符号长整型（针对 o、u 和 x），或双精度型（针对 e、f 和 g） L ：长双精度型（针对 e、f 和 g）
type	一个字符，指定了要被读取的数据类型以及数据读取方式。具体参见下一个表格。

fscanf 类型说明符：

类型	合格的输入	参数的类型
c	单个字符：读取下一个字符。如果指定了一个不为 1 的宽度 width，函数会读取 width 个字符，并通过参数传递，把它们存储在数组中连续位置。在末尾不会追加空字符。	char *
d	十进制整数：数字前面的 + 或 - 号是可选的。	int *
e,E,f,g,G	浮点数：包含了一个小数点、一个可选的前置符号 + 或 -、一个可选的后置字符 e 或 E，以及一个十进制数字。两个有效的实例 -732.103 和 7.12e4	float *
o	八进制整数。	int *
s	字符串。这将读取连续字符，直到遇到一个空格字符（空格字符可以是空白、换行和制表符）。	char *
u	无符号的十进制整数。	unsigned int *
x,X	十六进制整数。	int *

- 附加参数 -- 根据不同的 format 字符串，函数可能需要一系列的附加参数，每个参数包含了一个要被插入的值，替换了 format 参数中指定的每个 % 标签。参数的个数应与 % 标签的个数相同。

返回值

如果成功，该函数返回成功匹配和赋值的个数。如果到达文件末尾或发生读错误，则返回 EOF。

实例

下面的实例演示了 fscanf() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char str1[10], str2[10], str3[10];
   int year;
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fputs("We are in 2014", fp);

   rewind(fp);
   fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);

   printf("Read String1 |%s|\n", str1 );
   printf("Read String2 |%s|\n", str2 );
   printf("Read String3 |%s|\n", str3 );
   printf("Read Integer |%d|\n", year );

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Read String1 |We|
Read String2 |are|
Read String3 |in|
Read Integer |2014|
```

C 库函数 - scanf()

描述

C 库函数 **int scanf(const char *format, ...)** 从标准输入 stdin 读取格式化输入。

声明

下面是 scanf() 函数的声明。

```
int scanf(const char *format, ...)
```

参数

- **format** -- 这是 C 字符串，包含了以下各项中的一个或多个：空格字符、非空格字符 和 *format* 说明符。
  format 说明符形式为 **[=%[*][width][modifiers]type=]**，具体讲解如下：

参数	描述
*	这是一个可选的星号，表示数据是从流 stream 中读取的，但是可以被忽视，即它不存储在对应的参数中。
width	这指定了在当前读取操作中读取的最大字符数。
modifiers	为对应的附加参数所指向的数据指定一个不同于整型（针对 d、i 和 n）、无符号整型（针对 o、u 和 x）或浮点型（针对 e、f 和 g）的大小： h ：短整型（针对 d、i 和 n），或无符号短整型（针对 o、u 和 x） l ：长整型（针对 d、i 和 n），或无符号长整型（针对 o、u 和 x），或双精度型（针对 e、f 和 g） L ：长双精度型（针对 e、f 和 g）
type	一个字符，指定了要被读取的数据类型以及数据读取方式。具体参见下一个表格。

scanf 类型说明符：

类型	合格的输入	参数的类型
c	单个字符：读取下一个字符。如果指定了一个不为 1 的宽度 width，函数会读取 width 个字符，并通过参数传递，把它们存储在数组中连续位置。在末尾不会追加空字符。	char *
d	十进制整数：数字前面的 + 或 - 号是可选的。	int *
e,E,f,g,G	浮点数：包含了一个小数点、一个可选的前置符号 + 或 -、一个可选的后置字符 e 或 E，以及一个十进制数字。两个有效的实例 -732.103 和 7.12e4	float *
o	八进制整数。	int *
s	字符串。这将读取连续字符，直到遇到一个空格字符（空格字符可以是空白、换行和制表符）。	char *
u	无符号的十进制整数。	unsigned int *
x,X	十六进制整数。	int *

- 附加参数 -- 根据不同的 format 字符串，函数可能需要一系列的附加参数，每个参数包含了一个要被插入的值，替换了 format 参数中指定的每个 % 标签。参数的个数应与 % 标签的个数相同。

返回值

如果成功，该函数返回成功匹配和赋值的个数。如果到达文件末尾或发生读错误，则返回 EOF。

实例

下面的实例演示了 scanf() 函数的用法。

```
#include <stdio.h>

int main()
{
   char str1[20], str2[30];

   printf("请输入用户名：");
   scanf("%s", &str1);

   printf("请输入您的网站：");
   scanf("%s", &str2);

   printf("输入的用户名：%s\n", str1);
   printf("输入的网站：%s", str2);

   return(0);
}
```

让我们编译并运行上面的程序，这将在交互模式下产生以下结果：

```
请输入用户名：admin
请输入您的网站：www.w3cschool.cc

输入的用户名：admin
输入的网站：www.w3cschool.cc
```

C 库函数 - sscanf()

描述

C 库函数 **int sscanf(const char *str, const char *format, ...)** 从字符串读取格式化输入。

声明

下面是 sscanf() 函数的声明。

```
int sscanf(const char *str, const char *format, ...)
```

参数

- **str** -- 这是 C 字符串，是函数检索数据的源。
- **format** -- 这是 C 字符串，包含了以下各项中的一个或多个：空格字符、非空格字符 和 *format* 说明符。
  format 说明符形式为 **[=%[*][width][modifiers]type=]**，具体讲解如下：

参数	描述
*	这是一个可选的星号，表示数据是从流 stream 中读取的，但是可以被忽视，即它不存储在对应的参数中。
width	这指定了在当前读取操作中读取的最大字符数。
modifiers	为对应的附加参数所指向的数据指定一个不同于整型（针对 d、i 和 n）、无符号整型（针对 o、u 和 x）或浮点型（针对 e、f 和 g）的大小： h ：短整型（针对 d、i 和 n），或无符号短整型（针对 o、u 和 x） l ：长整型（针对 d、i 和 n），或无符号长整型（针对 o、u 和 x），或双精度型（针对 e、f 和 g） L ：长双精度型（针对 e、f 和 g）
type	一个字符，指定了要被读取的数据类型以及数据读取方式。具体参见下一个表格。

sscanf 类型说明符：

类型	合格的输入	参数的类型
c	单个字符：读取下一个字符。如果指定了一个不为 1 的宽度 width，函数会读取 width 个字符，并通过参数传递，把它们存储在数组中连续位置。在末尾不会追加空字符。	char *
d	十进制整数：数字前面的 + 或 - 号是可选的。	int *
e,E,f,g,G	浮点数：包含了一个小数点、一个可选的前置符号 + 或 -、一个可选的后置字符 e 或 E，以及一个十进制数字。两个有效的实例 -732.103 和 7.12e4	float *
o	八进制整数。	int *
s	字符串。这将读取连续字符，直到遇到一个空格字符（空格字符可以是空白、换行和制表符）。	char *
u	无符号的十进制整数。	unsigned int *
x,X	十六进制整数。	int *

- 附加参数 -- 这个函数接受一系列的指针作为附加参数，每一个指针都指向一个对象，对象类型由 format 字符串中相应的 % 标签指定，参数与 % 标签的顺序相同。

  针对检索数据的 format 字符串中的每个 format 说明符，应指定一个附加参数。如果您想要把 sscanf 操作的结果存储在一个普通的变量中，您应该在标识符前放置引用运算符（&），例如：

```
int n;
sscanf (str,"%d",&amp;n);
```

返回值

如果成功，该函数返回成功匹配和赋值的个数。如果到达文件末尾或发生读错误，则返回 EOF。

实例

下面的实例演示了 sscanf() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int day, year;
   char weekday[20], month[20], dtm[100];

   strcpy( dtm, "Saturday March 25 1989" );
   sscanf( dtm, "%s %s %d  %d", weekday, month, &day, &year );

   printf("%s %d, %d = %s\n", month, day, year, weekday );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
March 25, 1989 = Saturday
```

C 库函数 - fgetc()

描述

C 库函数 **int fgetc(FILE *stream)** 从指定的流 stream 获取下一个字符（一个无符号字符），并把位置标识符往前移动。

声明

下面是 fgetc() 函数的声明。

```
int fgetc(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要在上面执行操作的流。

返回值

该函数以无符号 char 强制转换为 int 的形式返回读取的字符，如果到达文件末尾或发生读错误，则返回 EOF。

实例

下面的实例演示了 fgetc() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;
   int n = 0;

   fp = fopen("file.txt","r");
   if(fp == NULL)
   {
      perror("打开文件时发生错误");
      return(-1);
   }
   do
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }while(1);

   fclose(fp);
   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下。文件将作为实例中的输入：

```
We are in 2014
```

让我们编译并运行上面的程序，这将产生以下结果：

```
We are in 2014
```

C 库函数 - fgets()

描述

C 库函数 **char *fgets(char* str, int n, FILE *stream)** 从指定的流 stream 读取一行，并把它存储在 **str** 所指向的字符串内。当读取 **(n-1)** 个字符时，或者读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。

声明

下面是 fgets() 函数的声明。

```
char *fgets(char *str, int n, FILE *stream)
```

参数

- **str** -- 这是指向一个字符数组的指针，该数组存储了要读取的字符串。
- **n** -- 这是要读取的最大字符数（包括最后的空字符）。通常是使用以 str 传递的数组长度。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要从中读取字符的流。

返回值

如果成功，该函数返回相同的 str 参数。如果到达文件末尾或者没有读取到任何字符，str 的内容保持不变，并返回一个空指针。

如果发生错误，返回一个空指针。

实例

下面的实例演示了 fgets() 函数的用法。

```
#include <stdio.h>

int main()
{
   FILE *fp;
   char str[60];

   /* 打开用于读取的文件 */
   fp = fopen("file.txt" , "r");
   if(fp == NULL) {
      perror("打开文件时发生错误");
      return(-1);
   }
   if( fgets (str, 60, fp)!=NULL ) {
      /* 向标准输出 stdout 写入内容 */
      puts(str);
   }
   fclose(fp);

   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下。文件将作为实例中的输入：

```
We are in 2014
```

让我们编译并运行上面的程序，这将产生以下结果：

```
We are in 2014
```

C 库函数 - fputc()

描述

C 库函数 **int fputc(int char, FILE *stream)** 把参数 **char** 指定的字符（一个无符号字符）写入到指定的流 stream 中，并把位置标识符往前移动。

声明

下面是 fputc() 函数的声明。

```
int fputc(int char, FILE *stream)
```

参数

- **char** -- 这是要被写入的字符。该字符以其对应的 int 值进行传递。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要被写入字符的流。

返回值

如果没有发生错误，则返回被写入的字符。如果发生错误，则返回 EOF，并设置错误标识符。

实例

下面的实例演示了 fputc() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int ch;

   fp = fopen("file.txt", "w+");
   for( ch = 33 ; ch <= 100; ch++ )
   {
      fputc(ch, fp);
   }
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将在当前目录中创建文件 **file.txt**，它的内容如下：

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcd
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - fputs()

描述

C 库函数 **int fputs(const char** ***str, FILE*** **stream)** 把字符串写入到指定的流 stream 中，但不包括空字符。

声明

下面是 fputs() 函数的声明。

```
int fputs(const char *str, FILE *stream)
```

参数

- **str** -- 这是一个数组，包含了要写入的以空字符终止的字符序列。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要被写入字符串的流。

返回值

该函数返回一个非负值，如果发生错误则返回 EOF。

实例

下面的实例演示了 fputs() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   fp = fopen("file.txt", "w+");

   fputs("这是 C 语言。", fp);
   fputs("这是一种系统程序设计语言。", fp);

   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将创建文件 **file.txt**，它的内容如下：

```
这是 C 语言。这是一种系统程序设计语言。
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - getc()

描述

C 库函数 **int getc(FILE *stream)** 从指定的流 stream 获取下一个字符（一个无符号字符），并把位置标识符往前移动。

声明

下面是 getc() 函数的声明。

```
int getc(FILE *stream)
```

参数

- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要在上面执行操作的流。

返回值

该函数以无符号 char 强制转换为 int 的形式返回读取的字符，如果到达文件末尾或发生读错误，则返回 EOF。

实例

下面的实例演示了 getc() 函数的用法。

```
#include<stdio.h>

int main()
{
   char c;

   printf("请输入字符：");
   c = getc(stdin);
   printf("输入的字符：");
   putc(c, stdout);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
请输入字符：a
输入的字符：a
```

C 库函数 - getchar()

描述

C 库函数 **int getchar(void)** 从标准输入 stdin 获取一个字符（一个无符号字符）。这等同于 **getc** 带有 stdin 作为参数。

声明

下面是 getchar() 函数的声明。

```
int getchar(void)
```

参数

- **NA**

返回值

该函数以无符号 char 强制转换为 int 的形式返回读取的字符，如果到达文件末尾或发生读错误，则返回 EOF。

实例

下面的实例演示了 getchar() 函数的用法。

```
#include <stdio.h>

int main ()
{
   char c;

   printf("请输入字符：");
   c = getchar();

   printf("输入的字符：");
   putchar(c);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
请输入字符：a
输入的字符：a
```

C 库函数 - gets()

描述

C 库函数 **char *gets(char *str)** 从标准输入 stdin 读取一行，并把它存储在 str 所指向的字符串中。当读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。

声明

下面是 gets() 函数的声明。

```
char *gets(char *str)
```

参数

- **str** -- 这是指向一个字符数组的指针，该数组存储了 C 字符串。

返回值

如果成功，该函数返回 str。如果发生错误或者到达文件末尾时还未读取任何字符，则返回 NULL。

实例

下面的实例演示了 gets() 函数的用法。

```
#include <stdio.h>

int main()
{
   char str[50];

   printf("请输入一个字符串：");
   gets(str);

   printf("您输入的字符串是：%s", str);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
请输入一个字符串：w3cschool.cc
您输入的字符串是：w3cschool.cc
```

C 库函数 - putc()

描述

C 库函数 **int putc(int char, FILE *stream)** 把参数 **char** 指定的字符（一个无符号字符）写入到指定的流 stream 中，并把位置标识符往前移动。

声明

下面是 putc() 函数的声明。

```
int putc(int char, FILE *stream)
```

参数

- **char** -- 这是要被写入的字符。该字符以其对应的 int 值进行传递。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要被写入字符的流。

返回值

该函数以无符号 char 强制转换为 int 的形式返回写入的字符，如果发生错误则返回 EOF。

实例

下面的实例演示了 putc() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int ch;

   fp = fopen("file.txt", "w");
   for( ch = 33 ; ch <= 100; ch++ )
   {
      putc(ch, fp);
   }
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将在当前目录中创建文件 **file.txt**，它的内容如下：

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcd
```

现在让我们使用下面的程序查看上面文件的内容：

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1)
   {
      c = fgetc(fp);
      if( feof(fp) )
      {
          break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

C 库函数 - putchar()

描述

C 库函数 **int putchar(int char)** 把参数 char 指定的字符（一个无符号字符）写入到标准输出 stdout 中。

声明

下面是 putchar() 函数的声明。

```
int putchar(int char)
```

参数

- **char** -- 这是要被写入的字符。该字符以其对应的 int 值进行传递。

返回值

该函数以无符号 char 强制转换为 int 的形式返回写入的字符，如果发生错误则返回 EOF。

实例

下面的实例演示了 putchar() 函数的用法。

```
#include <stdio.h>

int main ()
{
   char ch;

   for(ch = 'A' ; ch <= 'Z' ; ch++) {
      putchar(ch);
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

C 库函数 - puts()

描述

C 库函数 **int puts(const char *str)** 把一个字符串写入到标准输出 stdout，直到空字符，但不包括空字符。换行符会被追加到输出中。

声明

下面是 puts() 函数的声明。

```
int puts(const char *str)
```

参数

- **str** -- 这是要被写入的 C 字符串。

返回值

如果成功，该函数返回一个非负值，如果发生错误则返回 EOF。

实例

下面的实例演示了 puts() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main()
{
   char str1[15];
   char str2[15];

   strcpy(str1, "w3cschool");
   strcpy(str2, "w3cbird");

   puts(str1);
   puts(str2);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
w3cschool
w3cbird
```

C 库函数 - ungetc()

描述

C 库函数 **int ungetc(int char, FILE *stream)** 把字符 **char**（一个无符号字符）推入到指定的流 **stream** 中，以便它是下一个被读取到的字符。

声明

下面是 ungetc() 函数的声明。

```
int ungetc(int char, FILE *stream)
```

参数

- **char** -- 这是要被推入的字符。该字符以其对应的 int 值进行传递。
- **stream** -- 这是指向 FILE 对象的指针，该 FILE 对象标识了输入流。

返回值

如果成功，则返回被推入的字符，否则返回 EOF，且流 stream 保持不变。

实例

下面的实例演示了 ungetc() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;
   int c;
   char buffer [256];

   fp = fopen("file.txt", "r");
   if( fp == NULL )
   {
      perror("打开文件时发生错误");
      return(-1);
   }
   while(!feof(fp))
   {
      c = getc (fp);
      /* 把 ! 替换为 + */
      if( c == '!' )
      {
         ungetc ('+', fp);
      }
      else
      {
         ungetc(c, fp);
      }
      fgets(buffer, 255, fp);
      fputs(buffer, stdout);
   }
   return(0);
}
```

假设我们有一个文本文件 **file.txt**，它的内容如下。文件将作为实例中的输入：

```
this is w3cschool
!c standard library
!library functions and macros
```

让我们编译并运行上面的程序，这将产生以下结果：

```
this is w3cschool
+c standard library
+library functions and macros
+library functions and macros
```

C 库函数 - perror()

描述

C 库函数 **void perror(const char *str)** 把一个描述性错误消息输出到标准错误 stderr。首先输出字符串 **str**，后跟一个冒号，然后是一个空格。

声明

下面是 perror() 函数的声明。

```
void perror(const char *str)
```

参数

- **str** -- 这是 C 字符串，包含了一个自定义消息，将显示在原本的错误消息之前。

返回值

该函数不返回任何值。

实例

下面的实例演示了 perror() 函数的用法。

```
#include <stdio.h>

int main ()
{
   FILE *fp;

   /* 首先重命名文件 */
   rename("file.txt", "newfile.txt");

   /* 现在让我们尝试打开相同的文件 */
   fp = fopen("file.txt", "r");
   if( fp == NULL ) {
      perror("Error: ");
      return(-1);
   }
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果，因为我们尝试打开一个不存在的文件：

```
Error: : No such file or directory
```

C 标准库 - <stdlib.h>

简介

stdlib .h 头文件定义了四个变量类型、一些宏和各种通用工具函数。

库变量

下面是头文件 stdlib.h 中定义的变量类型：

变量	描述
size_t	这是无符号整数类型，它是 **sizeof** 关键字的结果。
wchar_t	这是一个宽字符常量大小的整数类型。
div_t	这是 **div** 函数返回的结构。
ldiv_t	这是 **ldiv** 函数返回的结构。

库宏

下面是头文件 stdlib.h 中定义的宏：

宏	描述
NULL	这个宏是一个空指针常量的值。
EXIT_FAILURE	这是 exit 函数失败时要返回的值。
EXIT_SUCCESS	这是 exit 函数成功时要返回的值。
RAND_MAX	这个宏是 rand 函数返回的最大值。
MB_CUR_MAX	这个宏表示在多字节字符集中的最大字符数，不能大于 MB_LEN_MAX。

库函数

下面是头文件 stdlib.h 中定义的函数：

函数	描述
double atof(const char *str)	把参数 *str* 所指向的字符串转换为一个浮点数（类型为 double 型）。

int atoi(const char *str)	把参数 *str* 所指向的字符串转换为一个整数（类型为 int 型）。
long int atol(const char *str)	把参数 *str* 所指向的字符串转换为一个长整数（类型为 long int 型）。
double strtod(const char *str, char **endptr)	把参数 *str* 所指向的字符串转换为一个浮点数（类型为 double 型）。
long int strtol(const char *str, char **endptr, int base)	把参数 *str* 所指向的字符串转换为一个长整数（类型为 long int 型）。
unsigned long int strtoul(const char *str, char **endptr, int base)	把参数 *str* 所指向的字符串转换为一个无符号长整数（类型为 unsigned long int 型）。
void *calloc(size_t nitems, size_t size)	分配所需的内存空间，并返回一个指向它的指针。
void free(void *ptr	释放之前调用 *calloc*、*malloc* 或 *realloc* 所分配的内存空间。
void *malloc(size_t size)	分配所需的内存空间，并返回一个指向它的指针。
void *realloc(void *ptr, size_t size)	尝试重新调整之前调用 *malloc* or *calloc* 所分配的 ptr 所指向的内存块的大小。
void abort(void)	使一个异常程序终止。
int atexit(void (*func)(void))	当程序正常终止时，调用指定的函数 **func**。
void exit(int status)	是程序正常终止。
char *getenv(const char *name)	搜索 name 所指向的环境字符串，并返回相关的值给字符串。
int system(const char *string)	由 string 指定的命令传给要被命令处理器执行的主机环境。
void *bsearch(const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *))	执行二进制搜索。
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))	数组排序。
int abs(int x)	返回 x 的绝对值。
div_t div(int numer, int denom)	分子除以分母。
long int labs(long int x)	返回 x 的绝对值。
ldiv_t ldiv(long int numer, long int denom)	分子除以分母。
int rand(void)	返回一个范围在 0 到 *RAND_MAX* 之间的伪随机数。

void srand(unsigned int seed)	该函数播种由函数 **rand** 使用的随机数发生器。
int mblen(const char *str, size_t n)	返回参数 *str* 所指向的多字节字符的长度。
size_t mbstowcs(schar_t *pwcs, const char *str, size_t n)	把参数 *str* 所指向的多字节字符的字符串转换为参数 *pwcs* 所指向的数组。
int mbtowc(whcar_t *pwc, const char *str, size_t n)	检查参数 *str* 所指向的多字节字符。
size_t wcstombs(char *str, const wchar_t *pwcs, size_t n)	把数组 *pwcs* 中存储的编码转换为多字节字符，并把它们存储在字符串 *str* 中。
int wctomb(char *str, wchar_t wchar)	检查对应于参数 *wchar* 所给出的多字节字符的编码。

C 库函数 - atof()

描述

C 库函数 **double atof(const char *str)** 把参数 **str** 所指向的字符串转换为一个浮点数（类型为 double 型）。

声明

下面是 atof() 函数的声明。

```
double atof(const char *str)
```

参数

- **str** -- 要转换为浮点数的字符串。

返回值

函数返回转换后的双精度浮点数，如果没有执行有效的转换，则返回零（0.0）。

实例

下面的实例演示了 atof() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   float val;
   char str[20];

   strcpy(str, "98993489");
   val = atof(str);
   printf("字符串值 = %s, 浮点值 = %f\n", str, val);

   strcpy(str, "w3cschool.cc");
   val = atof(str);
   printf("字符串值 = %s, 浮点值 = %f\n", str, val);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
字符串值 = 98993489, 浮点值 = 98993488.000000
字符串值 = w3cschool.cc, 浮点值 = 0.000000
```

C 库函数 - atoi()

描述

C 库函数 **int atoi(const char *str)** 把参数 **str** 所指向的字符串转换为一个整数（类型为 int 型）。

声明

下面是 atoi() 函数的声明。

```
int atoi(const char *str)
```

参数

- **str** -- 要转换为整数的字符串。

返回值

该函数返回转换后的长整数，如果没有执行有效的转换，则返回零。

实例

下面的实例演示了 atoi() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   int val;
   char str[20];

   strcpy(str, "98993489");
   val = atoi(str);
   printf("字符串值 = %s, 整型值 = %d\n", str, val);

   strcpy(str, "w3cschool.cc");
   val = atoi(str);
   printf("字符串值 = %s, 整型值 = %d\n", str, val);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
字符串值 = 98993489, 整型值 = 98993489
字符串值 = w3cschool.cc, 整型值 = 0
```

C 库函数 - atol()

描述

C 库函数 **long int atol(const char *str)** 把参数 **str** 所指向的字符串转换为一个长整数（类型为 long int 型）。

声明

下面是 atol() 函数的声明。

```
long int atol(const char *str)
```

参数

- **str** -- 要转换为长整数的字符串。

返回值

该函数返回转换后的长整数，如果没有执行有效的转换，则返回零。

实例

下面的实例演示了 atol() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   long val;
   char str[20];

   strcpy(str, "98993489");
   val = atol(str);
   printf("字符串值 = %s, 长整型值 = %ld\n", str, val);

   strcpy(str, "w3cschool.cc");
   val = atol(str);
   printf("字符串值 = %s, 长整型值 = %ld\n", str, val);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
字符串值 = 98993489, 长整型值 = 98993489
字符串值 = w3cschool.cc, 长整型值 = 0
```

C 库函数 - strtod()

描述

C 库函数 **double strtod(const char *str, char** endptr) 把参数 str 所指向的字符串转换为一个浮点数（类型为 **double** 型）。如果 endptr** 不为空，则指向转换中最后一个字符后的字符的指针会存储在 endptr 引用的位置。

声明

下面是 strtod() 函数的声明。

```
double strtod(const char *str, char **endptr)
```

参数

- **str** -- 要转换为双精度浮点数的字符串。
- **endptr** -- 对类型为 char 的对象的引用，其值由函数设置为 *str* 中数值后的下一个字符。

返回值

该函数返回转换后的双精度浮点数，如果没有执行有效的转换，则返回零（0.0）。

实例

下面的实例演示了 strtod() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  char str[30] = "20.30300 This is test";
   char *ptr;
   double ret;

   ret = strtod(str, &ptr);
   printf("数字（double）是 %lf\n", ret);
   printf("字符串部分是 |%s|", ptr);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
数字（double）是 20.303000
字符串部分是 | This is test|
```

C 库函数 - strtol()

描述

C 库函数 **long int strtol(const char *str, char** endptr, int base) 把参数 str 所指向的字符串根据给定的 base** 转换为一个长整数（类型为 long int 型），base 必须介于 2 和 36（包含）之间，或者是特殊值 0。

声明

下面是 strtol() 函数的声明。

```
long int strtol(const char *str, char **endptr, int base)
```

参数

- **str** -- 要转换为长整数的字符串。
- **endptr** -- 对类型为 char 的对象的引用，其值由函数设置为 **str* 中数值后的下一个字符。
- **base** -- 基数，必须介于 2 和 36（包含）之间，或者是特殊值 0。

返回值

该函数返回转换后的长整数，如果没有执行有效的转换，则返回一个零值。

实例

下面的实例演示了 strtol() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char str[30] = "2030300 This is test";
   char *ptr;
   long ret;

   ret = strtol(str, &ptr, 10);
   printf("数字（无符号长整数）是 %ld\n", ret);
   printf("字符串部分是 |%s|", ptr);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
数字（无符号长整数）是 2030300
字符串部分是 | This is test|
```

C 库函数 - strtoul()

描述

C 库函数 **unsigned long int strtoul(const char *str, char** endptr, int base) 把参数 str 所指向的字符串根据给定的 base** 转换为一个无符号长整数（类型为 unsigned long int 型），base 必须介于 2 和 36（包含）之间，或者是特殊值 0。

声明

下面是 strtoul() 函数的声明。

```
unsigned long int strtoul(const char *str, char **endptr, int base)
```

参数

- **str** -- 要转换为无符号长整数的字符串。
- **endptr** -- 对类型为 char* 的对象的引用，其值由函数设置为 str 中数值后的下一个字符。
- **base** -- 基数，必须介于 2 和 36（包含）之间，或者是特殊值 0。

返回值

该函数返回转换后的长整数，如果没有执行有效的转换，则返回一个零值。

实例

下面的实例演示了 strtoul() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char str[30] = "2030300 This is test";
   char *ptr;
   long ret;

   ret = strtoul(str, &ptr, 10);
   printf("数字（无符号长整数）是 %lu\n", ret);
   printf("字符串部分是 |%s|", ptr);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
数字（无符号长整数）是 2030300
字符串部分是 | This is test|
```

C 库函数 - calloc()

描述

C 库函数 **void *calloc(size_t nitems, size_t size)** 分配所需的内存空间，并返回一个指向它的指针。**malloc** 和 **calloc** 之间的不同点是，malloc 不会设置内存为零，而 calloc 会设置分配的内存为零。

声明

下面是 calloc() 函数的声明。

```
void *calloc(size_t nitems, size_t size)
```

参数

- **nitems** -- 要被分配的元素个数。
- **size** -- 元素的大小。

返回值

该函数返回一个指针，指向已分配的内存。如果请求失败，则返回 NULL。

实例

下面的实例演示了 calloc() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i, n;
   int *a;

   printf("要输入的元素个数：");
   scanf("%d",&n);

   a = (int*)calloc(n, sizeof(int));
   printf("输入 %d 个数字：\n",n);
   for( i=0 ; i < n ; i++ )
   {
      scanf("%d",&a[i]);
   }

   printf("输入的数字为：");
   for( i=0 ; i < n ; i++ ) {
      printf("%d ",a[i]);
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
要输入的元素个数：3
输入 3 个数字：
22
55
14
输入的数字为：22 55 14
```

C 库函数 - free()

描述

C 库函数 **void free(void *ptr)** 释放之前调用 calloc、malloc 或 realloc 所分配的内存空间。

声明

下面是 free() 函数的声明。

```
void free(void *ptr)
```

参数

- **ptr** -- 指针指向一个要释放内存的内存块，该内存块之前是通过调用 malloc、calloc 或 realloc 进行分配内存的。如果传递的参数是一个空指针，则不会执行任何动作。

返回值

该函数不返回任何值。

实例

下面的实例演示了 free() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char *str;

   /* 最初的内存分配 */
   str = (char *) malloc(15);
   strcpy(str, "w3cschool");
   printf("String = %s,  Address = %u\n", str, str);

   /* 重新分配内存 */
   str = (char *) realloc(str, 25);
   strcat(str, ".cc");
   printf("String = %s,  Address = %u\n", str, str);

   /* 释放已分配的内存 */
   free(str);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
String = w3cschool, Address = 355090448
String = w3cschool.cc, Address = 355090448
```

C 库函数 - malloc()

描述

C 库函数 **void *malloc(size_t size)** 分配所需的内存空间，并返回一个指向它的指针。

声明

下面是 malloc() 函数的声明。

```
void *malloc(size_t size)
```

参数

- **size** -- 内存块的大小，以字节为单位。

返回值

该函数返回一个指针 ，指向已分配大小的内存。如果请求失败，则返回 NULL。

实例

下面的实例演示了 malloc() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char *str;

   /* 最初的内存分配 */
   str = (char *) malloc(15);
   strcpy(str, "w3cschool");
   printf("String = %s,  Address = %u\n", str, str);

   /* 重新分配内存 */
   str = (char *) realloc(str, 25);
   strcat(str, ".cc");
   printf("String = %s,  Address = %u\n", str, str);

   free(str);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
String = w3cschool, Address = 355090448
String = w3cschool.cc, Address = 355090448
```

C 库函数 - realloc()

描述

C 库函数 **void *realloc(void* ptr, size_t size)** 尝试重新调整之前调用 **malloc** 或 **calloc** 所分配的 **ptr** 所指向的内存块的大小。

声明

下面是 realloc() 函数的声明。

```
void *realloc(void *ptr, size_t size)
```

参数

- **ptr** -- 指针指向一个要重新分配内存的内存块，该内存块之前是通过调用 malloc、calloc 或 realloc 进行分配内存的。如果为空指针，则会分配一个新的内存块，且函数返回一个指向它的指针。
- **size** -- 内存块的新的大小，以字节为单位。如果大小为 0，且 ptr 指向一个已存在的内存块，则 ptr 所指向的内存块会被释放，并返回一个空指针。

返回值

该函数返回一个指针 ，指向重新分配大小的内存。如果请求失败，则返回 NULL。

实例

下面的实例演示了 realloc() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char *str;

   /* 最初的内存分配 */
   str = (char *) malloc(15);
   strcpy(str, "w3cschool");
   printf("String = %s,  Address = %u\n", str, str);

   /* 重新分配内存 */
   str = (char *) realloc(str, 25);
   strcat(str, ".cc");
   printf("String = %s,  Address = %u\n", str, str);

   free(str);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
String = w3cschool, Address = 355090448
String = w3cschool.cc, Address = 355090448
```

C 库函数 - abort()

描述

C 库函数 **void abort(void)** 中止程序执行，直接从调用的地方跳出。

声明

下面是 abort() 函数的声明。

```
void abort(void)
```

参数

- **NA**

返回值

该函数不返回任何值。

实例

下面的实例演示了 abort() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   FILE *fp;

   printf("准备打开 nofile.txt\n");
   fp = fopen( "nofile.txt","r" );
   if(fp == NULL)
   {
      printf("准备终止程序\n");
      abort();
   }
   printf("准备关闭 nofile.txt\n");
   fclose(fp);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果，因为我们尝试打开的文件 **nofile.txt** 是不存在的：

```
准备打开 nofile.txt
准备终止程序
```

C 库函数 - atexit()

描述

C 库函数 **int atexit(void (*func)(void))** 当程序正常终止时，调用指定的函数 **func**。您可以在任何地方注册你的终止函数，但它会在程序终止的时候被调用。

声明

下面是 atexit() 函数的声明。

```
int atexit(void (*func)(void))
```

参数

- **func** -- 在程序终止时被调用的函数。

返回值

如果函数成功注册，则该函数返回零，否则返回一个非零值。

实例

下面的实例演示了 atexit() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

void functionA ()
{
   printf("这是函数A\n");
}

int main ()
{
   /* 注册终止函数 */
   atexit(functionA );

   printf("启动主程序...\n");

   printf("退出主程序...\n");

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
启动主程序...
退出主程序...
这是函数A
```

C 库函数 - exit()

描述

C 库函数 **void exit(int status)** 立即终止调用进程。任何属于该进程的打开的文件描述符都会被关闭，该进程的子进程由进程 1 继承，初始化，且会向父进程发送一个 SIGCHLD 信号。

声明

下面是 exit() 函数的声明。

```
void exit(int status)
```

参数

- **status** -- 返回给父进程的状态值。

返回值

该函数不返回值。

实例

下面的实例演示了 exit() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   printf("程序的开头....\n");

   printf("退出程序....\n");
   exit(0);

   printf("程序的结尾....\n");

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
程序的开头....
退出程序....
```

C 库函数 - getenv()

描述

C 库函数 **char *getenv(const char *name)** 搜索 name 所指向的环境字符串，并返回相关的值给字符串。

声明

下面是 getenv() 函数的声明。

```
char *getenv(const char *name)
```

参数

- **name** -- 包含被请求变量名称的 C 字符串。

返回值

该函数返回一个以 null 结尾的字符串，该字符串为被请求环境变量的值。如果该环境变量不存在，则返回 NULL。

实例

下面的实例演示了 getenv() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   printf("PATH : %s\n", getenv("PATH"));
   printf("HOME : %s\n", getenv("HOME"));
   printf("ROOT : %s\n", getenv("ROOT"));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
PATH : /sbin:/usr/sbin:/bin:/usr/bin:/usr/local/bin
HOME : /
ROOT : (null)
```

C 库函数 - system()

描述

C 库函数 **int system(const char *command)** 把 **command** 指定的命令名称或程序名称传给要被命令处理器执行的主机环境，并在命令完成后返回。

声明

下面是 system() 函数的声明。

```
int system(const char *command)
```

参数

- **command** -- 包含被请求变量名称的 C 字符串。

返回值

如果发生错误，则返回值为 -1，否则返回命令的状态。

实例

下面的实例演示了 system() 函数的用法，列出了 unix 机上当前目录下所有的文件和目录。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char command[50];

   strcpy( command, "ls -l" );
   system(command);

   return(0);
}
```

让我们编译并运行上面的程序，在 unix 机上将产生以下结果：

```
drwxr-xr-x 2 apache apache 4096 Aug 22 07:25 hsperfdata_apache
drwxr-xr-x 2 railo railo 4096 Aug 21 18:48 hsperfdata_railo
rw------ 1 apache apache 8 Aug 21 18:48 mod_mono_dashboard_XXGLOBAL_1
rw------ 1 apache apache 8 Aug 21 18:48 mod_mono_dashboard_asp_2
srwx---- 1 apache apache 0 Aug 22 05:28 mod_mono_server_asp
rw------ 1 apache apache 0 Aug 22 05:28 mod_mono_server_asp_1280495620
srwx---- 1 apache apache 0 Aug 21 18:48 mod_mono_server_global
```

下面的实例演示了 system() 函数的用法，列出了 windows 机上当前目录下所有的文件和目录。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char command[50];

   strcpy( command, "dir" );
   system(command);

   return(0);
}
```

让我们编译并运行上面的程序，在 windows 机上将产生以下结果：

```
a.txt
amit.doc
sachin
saurav
file.c
```

C 库函数 - bsearch()

描述

C 库函数 **void *bsearch(const void* key, const void *base, size_t nitems, size_t size, int* (*compar)(const void *, const void *))** 对 **nitems** 对象的数组执行二分查找，**base** 指向进行查找的数组，**key** 指向要查找的元素，**size** 指定数组中每个元素的大小。

数组的内容应根据 **compar** 所对应的比较函数升序排序。

声明

下面是 bsearch() 函数的声明。

```
void *bsearch(const void *key, const void *base, size_t nitems, size_t size, int (*compar
```

参数

- **key** -- 指向要查找的元素的指针，类型转换为 void*。
- **base** -- 指向进行查找的数组的第一个对象的指针，类型转换为 void*。
- **nitems** -- base 所指向的数组中元素的个数。
- **size** -- 数组中每个元素的大小，以字节为单位。
- **compar** -- 用来比较两个元素的函数。

返回值

如果查找成功，该函数返回一个指向数组中匹配元素的指针，否则返回空指针。.

实例

下面的实例演示了 bsearch() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int cmpfunc(const void * a, const void * b)
{
   return ( *(int*)a - *(int*)b );
}

int values[] = { 5, 20, 29, 32, 63 };

int main ()
{
   int *item;
   int key = 32;

   /* 使用 bsearch() 在数组中查找值 32 */
   item = (int*) bsearch (&key, values, 5, sizeof (int), cmpfunc);
   if( item != NULL )
   {
      printf("Found item = %d\n", *item);
   }
   else
   {
      printf("Item = %d could not be found\n", *item);
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Found item = 32
```

C 库函数 - qsort()

描述

C 库函数 **void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))** 对数组进行排序。

声明

下面是 qsort() 函数的声明。

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void
```

参数

- **base** -- 指向要排序的数组的第一个元素的指针。
- **nitems** -- 由 base 指向的数组中元素的个数。
- **size** -- 数组中每个元素的大小，以字节为单位。
- **compar** -- 用来比较两个元素的函数。

返回值

该函数不返回任何值。

实例

下面的实例演示了 qsort() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int values[] = { 88, 56, 100, 2, 25 };

int cmpfunc (const void * a, const void * b)
{
   return ( *(int*)a - *(int*)b );
}

int main()
{
   int n;

   printf("排序之前的列表：\n");
   for( n = 0 ; n < 5; n++ ) {
      printf("%d ", values[n]);
   }

   qsort(values, 5, sizeof(int), cmpfunc);

   printf("\n排序之后的列表：\n");
   for( n = 0 ; n < 5; n++ ) {
      printf("%d ", values[n]);
   }

  return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
排序之前的列表：
88 56 100 2 25
排序之后的列表：
2 25 56 88 100
```

C 库函数 - abs()

描述

C 库函数 **int abs(int x)** 返回 **x** 的绝对值。

声明

下面是 abs() 函数的声明。

```
int abs(int x)
```

参数

- **x** -- 完整的值。

返回值

该函数返回 x 的绝对值。

实例

下面的实例演示了 abs() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   int a, b;

   a = abs(5);
   printf("a 的值 = %d\n", a);

   b = abs(-10);
   printf("b 的值 = %d\n", b);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
a 的值 = 5
b 的值 = 10
```

C 库函数 - div()

描述

C 库函数 **div_t div(int numer, int denom)** 把 **numer**（分子）除以 **denom**（分母）。

声明

下面是 div() 函数的声明。

```
div_t div(int numer, int denom)
```

参数

- **numer** -- 分子。
- **denom** -- 分母。

返回值

该函数返回定义在 <cstdlib> 中的结构中的值，该结构有两个成员，如 div*t:_int quot; int rem;*。

实例

下面的实例演示了 div() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   div_t output;

   output = div(27, 4);
   printf("(27/ 4) 的商  = %d\n", output.quot);
   printf("(27/4) 的余数 = %d\n", output.rem);

   output = div(27, 3);
   printf("(27/ 3) 的商 = %d\n", output.quot);
   printf("(27/3) 的余数 = %d\n", output.rem);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
(27/ 4) 的商 = 6
(27/4) 的余数 = 3
(27/ 3) 的商 = 9
(27/3) 的余数 = 0
```

C 库函数 - labs()

描述

C 库函数 **long int labs(long int x)** 返回 **x** 的绝对值。

声明

下面是 labs() 函数的声明。

```
long int labs(long int x)
```

参数

- **x** -- 完整的值。

返回值

该函数返回 x 的绝对值。

实例

下面的实例演示了 labs() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   long int a,b;

   a = labs(65987L);
   printf("a 的值 = %ld\n", a);

   b = labs(-1005090L);
   printf("b 的值 = %ld\n", b);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
a 的值 = 65987
b 的值 = 1005090
```

C 库函数 - ldiv()

描述

C 库函数 **div_t div(long int numer, long int denom)** 把 **numer**（分子）除以 **denom**（分母）。

声明

下面是 ldiv() 函数的声明。

```
div_t div(long int numer, long int denom)
```

参数

- **numer** -- 分子。
- **denom** -- 分母。

返回值

该函数返回定义在 <cstdlib> 中的结构中的值，该结构有两个成员，如 ldiv*t:_long quot; long rem;*。

实例

下面的实例演示了 ldiv() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   ldiv_t output;

   output = ldiv(100000L, 30000L);

   printf("商 = %ld\n", output.quot);

   printf("余数 = %ld\n", output.rem);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
商 = 3
余数 = 10000
```

C 库函数 - rand()

描述

C 库函数 **int rand(void)** 返回一个范围在 0 到 *RAND_MAX* 之间的伪随机数。

RAND_MAX 是一个常量，它的默认值在不同的实现中会有所不同，但是值至少是 32767。

声明

下面是 rand() 函数的声明。

```
int rand(void)
```

参数

- **NA**

返回值

该函数返回一个范围在 0 到 RAND_MAX 之间的整数值。

实例

下面的实例演示了 rand() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i, n;
   time_t t;

   n = 5;

   /* 初始化随机数发生器 */
   srand((unsigned) time(&t));

   /* 输出 0 到 49 之间的 5 个随机数 */
   for( i = 0 ; i < n ; i++ ) {
      printf("%d\n", rand() % 50);
   }

  return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
38
45
29
29
47
```

C 库函数 - srand()

描述

C 库函数 **void srand(unsigned int seed)** 播种由函数 **rand** 使用的随机数发生器。

声明

下面是 srand() 函数的声明。

```
void srand(unsigned int seed)
```

参数

- **seed** -- 这是一个整型值，用于伪随机数生成算法播种。

返回值

该函数不返回任何值。

实例

下面的实例演示了 srand() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
   int i, n;
   time_t t;

   n = 5;

   /* 初始化随机数发生器 */
   srand((unsigned) time(&t));

   /* 输出 0 到 50 之间的 5 个随机数 */
   for( i = 0 ; i < n ; i++ ) {
      printf("%d\n", rand() % 50);
   }

  return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
38
45
29
29
47
```

C 库函数 - mblen()

描述

C 库函数 **int mblen(const char *str, size_t n)** 返回参数 **str** 所指向的多字节字符的长度。

声明

下面是 mblen() 函数的声明。

```
int mblen(const char *str, size_t n)
```

参数

- **str** -- 指向多字节字符的第一个字节的指针。
- **n** -- 要检查的字符长度的最大字节数。

返回值

如果识别了一个非空宽字符，mblen() 函数返回 str 开始的多字节序列解析的字节数。如果识别了一个空宽字符，则返回 0。如果识别了一个无效的多字节序列，或者不能解析一个完整的多字节字符，则返回 -1。

实例

下面的实例演示了 mblen() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   int len;
   char *pmbnull  = NULL;
   char *pmb = (char *)malloc( MB_CUR_MAX );
   wchar_t *pwc = L"Hi";
   wchar_t *pwcs = (wchar_t *)malloc( sizeof( wchar_t ));

   printf("转换为多字节字符串\n");
   len = wcstombs( pmb, pwc, MB_CUR_MAX);
   printf("被转换的字符 %d\n", len);
   printf("第一个多字节字符的十六进制值：%#.4x\n", pmb);

   len = mblen( pmb, MB_CUR_MAX );
   printf( "多字节字符 %x 的字节长度：%u\n", pmb, len );

   pmb = NULL;

   len = mblen( pmb, MB_CUR_MAX );
   printf( "多字节字符 %x 的字节长度：%u\n", pmb, len );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
转换为多字节字符串
被转换的字符 1
第一个多字节字符的十六进制值：0x168c6010
多字节字符 168c6010 的字节长度：1
多字节字符 0 的字节长度：0
```

C 库函数 - mbstowcs()

描述

C 库函数 **size_t mbstowcs(schar_t *pwcs, const char* str, size_t n)** 把参数 **str** 所指向的多字节字符的字符串转换为参数 **pwcs** 所指向的数组。

声明

下面是 mbstowcs() 函数的声明。

```
size_t mbstowcs(schar_t *pwcs, const char *str, size_t n)
```

参数

- **pwcs** -- 指向一个 wchar_t 元素的数组，数组长度足以存储一个最大字符长度的宽字符串。
- **str** -- 要被转换的多字节字符字符串。
- **n** -- 要被转换的最大字符数。

返回值

该函数返回转换的字符数，不包括结尾的空字符。如果遇到一个无效的多字节字符，则返回 -1 值。

实例

下面的实例演示了 mbstowcs() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   int len;
   char *pmbnull  = NULL;
   char *pmb = (char *)malloc( MB_CUR_MAX );
   wchar_t *pwc = L"Hi";
   wchar_t *pwcs = (wchar_t *)malloc( sizeof( wchar_t ));

   printf("转换为多字节字符串\n");
   len = wcstombs( pmb, pwc, MB_CUR_MAX);
   printf("被转换的字符 %d\n", len);
   printf("第一个多字节字符的十六进制值：%#.4x\n", pmb);

   printf("转换回宽字符字符串\n");
   len = mbstowcs( pwcs, pmb, MB_CUR_MAX);
   printf("被转换的字符 %d\n", len);
   printf("第一个宽字符的十六进制值：%#.4x\n\n", pwcs);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
转换为多字节字符串
被转换的字符 1
第一个多字节字符的十六进制值：0x19a60010
转换回宽字符字符串
被转换的字符 1
第一个宽字符的十六进制值：0x19a60030
```

C 库函数 - mbtowc()

描述

C 库函数 **int mbtowc(whcar_t *pwc, const char* str, size_t n)** 把一个多字节序列转换为一个宽字符。

声明

下面是 mbtowc() 函数的声明。

```
int mbtowc(whcar_t *pwc, const char *str, size_t n)
```

参数

- **pwc** -- 指向类型为 wchar_t 对象的指针。
- **str** -- 指向多字节字符的第一个字节的指针。
- **n** -- 要被检查的最大字节数。

返回值

- 如果 str 不为 NULL，mbtowc() 函数返回 str 开始消耗的字节数，如果指向一个空字节，则返回 0，如果操作失败，则返回 -1。
- 如果 str 为 NULL，如果编码具有移位状态，则 mbtowc() 函数返回非零，如果编码是无状态的，则返回零。

实例

下面的实例演示了 mbtowc() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
   char *str = "这里是 w3cschool.cc";
   wchar_t mb[100];
   int len;

   len = mblen(NULL, MB_CUR_MAX);

   mbtowc(mb, str, len*strlen(str) );

   wprintf(L"%ls \n", mb );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果，因为它要以多字节形式输出结果，这是一种二进制输出。

```
???
```

C 库函数 - wcstombs()

描述

C 库函数 **size_t wcstombs(char *str, const wchar_t* pwcs, size_t n)** 把宽字符字符串 **pwcs** 转换为一个 **str** 开始的多字节字符串。最多会有 **n** 个字节被写入 **str** 中。

声明

下面是 wcstombs() 函数的声明。

```
size_t wcstombs(char *str, const wchar_t *pwcs, size_t n)
```

参数

- **str** -- 指向一个 char 元素的数组，至少有 n 字节长。
- **pwcs** -- 要被转换的宽字符字符串。
- **n** -- 要被写入到 str 中的最大字节数。

返回值

该函数返回转换和写入到 str 中的字节数，不包括结尾的空字符。如果遇到一个无效的多字节字符，则返回 -1 值。

实例

下面的实例演示了 wcstombs() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE 50

int main()
{
   size_t ret;
   char *MB = (char *)malloc( BUFFER_SIZE );
   wchar_t *WC = L"http://www.w3cschool.cc";

   /* 转换宽字符字符串 */
   ret = wcstombs(MB, WC, BUFFER_SIZE);

   printf("要转换的字符数 = %u\n", ret);
   printf("多字节字符 = %s\n\n", MB);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
要转换的字符数 = 23
多字节字符 = http://www.w3cschool.cc
```

C 库函数 - wctomb()

描述

C 库函数 **int wctomb(char *str, wchar_t wchar)** 把宽字符 **wchar** 转换为它的多字节表示形式，并把它存储在 **str** 指向的字符数组的开头。

声明

下面是 wctomb() 函数的声明。

```
int wctomb(char *str, wchar_t wchar)
```

参数

- **str** -- 一个指针，指向一个足以存储多字节字符的数组。
- **wchar** -- 类型为 wchar_t 的宽字符。

返回值

- 如果 str 不为 NULL，wctomb() 函数返回写入字节数组中的字节数。如果 wchar 不能被表示为一个多字节序列，则会返回 -1。
- 如果 str 为 NULL，如果编码具有移位状态，则 wctomb() 函数返回非零，如果编码是无状态的，则返回零。

实例

下面的实例演示了 wctomb() 函数的用法。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i;
   wchar_t wc = L'a';
   char *pmbnull = NULL;
   char *pmb = (char *)malloc(sizeof( char ));

   printf("要转换的宽字符：\n");
   i = wctomb( pmb, wc );
   printf("被转换的字符：%u\n", i);
   printf("多字节字符：%.1s\n", pmb);

   printf("当要转换的字符为 NULL 时尝试转换：\n");
   i = wctomb( pmbnull, wc );
   printf("被转换的字符：%u\n", i);
   /* 不会输出任何值 */
   printf("多字节字符：%.1s\n", pmbnull);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
要转换的宽字符：
被转换的字符：1
多字节字符：a
当要转换的字符为 NULL 时尝试转换：
被转换的字符：0
多字节字符：
```

C 标准库 - <string.h>

简介

string .h 头文件定义了一个变量类型、一个宏和各种操作字符数组的函数。

库变量

下面是头文件 string.h 中定义的变量类型：

变量	描述
size_t	这是无符号整数类型，它是 **sizeof** 关键字的结果。

库宏

下面是头文件 string.h 中定义的宏：

宏	描述
NULL	这个宏是一个空指针常量的值。

库函数

下面是头文件 string.h 中定义的函数：

函数	描述
void *memchr(const void *str, int c, size_t n)	在参数 *str* 所指向的字符串的前 n 个字节中搜索第一次出现字符 c（一个无符号字符）的位置。
int memcmp(const void *str1, const void *str2, size_t n)	把 *str1* 和 *str2* 的前 n 个字节进行比较。
void *memcpy(void *dest, const void *src, size_t n)	从 src 复制 n 个字符到 *dest*。
void	

*memmove(void *dest, const void *src, size_t n)	另一个用于从 *str2* 复制 n 个字符到 *str1* 的函数。
void *memset(void *str, int c, size_t n)	复制字符 c（一个无符号字符）到参数 *str* 所指向的字符串的前 n 个字符。
char *strcat(char *dest, const char *src)	把 *src* 所指向的字符串追加到 *dest* 所指向的字符串的结尾。
char *strncat(char *dest, const char *src, size_t n)	把 *src* 所指向的字符串追加到 *dest* 所指向的字符串的结尾，直到 n 字符长度为止。
char *strchr(const char *str, int c)	在参数 *str* 所指向的字符串中搜索第一次出现字符 c（一个无符号字符）的位置。
int strcmp(const char *str1, const char *str2)	把 *str1* 所指向的字符串和 *str2* 所指向的字符串进行比较。
int strncmp(const char *str1, const char *str2, size_t n)	把 *str1* 和 *str2* 进行比较，最多比较前 n 个字节。
int strcoll(const char *str1, const char *str2)	把 *str1* 和 *str2* 进行比较，结果取决于 LC_COLLATE 的位置设置。
char *strcpy(char *dest, const char *src)	把 *src* 所指向的字符串复制到 *dest*。
char *strncpy(char *dest, const char *src, size_t n)	把 *src* 所指向的字符串复制到 *dest*，最多复制 n 个字符。
size_t strcspn(const char *str1, const char *str2)	检索字符串 str1 开头连续有几个字符都不含字符串 str2 中的字符。
char *strerror(int errnum)	从内部数组中搜索错误号 errnum，并返回一个指向错误消息字符串的指针。
size_t strlen(const char *str)	计算字符串 str 的长度，直到空结束字符串，但不包括空结束字符串。
char *strpbrk(const char *str1, const char *str2)	检索字符串 *str1* 中匹配字符串 *str2* 中所指定的字符的第一个字符。也就是说，依次检验字符串s1中的字符，当被检验字符在字符串s2中也包含时，则停止检验，并返回该字符位置。
char *strrchr(const	在参数 *str* 所指向的字符串中搜索最后一次出现字符 c（一个无符

char *str, int c)	号字符）的位置。
size_t strspn(const char *str1, const char *str2)	检索字符串 *str1* 开头连续有几个字符都不含字符串 *str2* 中的字符。
char *strstr(const char *haystack, const char *needle)	在字符串 *haystack* 中查找第一次出现字符串 *needle*（不包含空结束字符串）的位置。
char *strtok(char *str, const char *delim)	分解字符串 *str* 为一组字符串，*delim* 为分隔符。
size_t strxfrm(char *dest, const char *src, size_t n)	根据程序当前的区域选项中的 LC_COLLATE 来转换字符串 **src** 的前 **n** 个字符，并把它们放置在字符串 **dest** 中。

C 库函数 - memchr()

描述

C 库函数 **void *memchr(const void* str, int c, size_t n)** 在参数 **str** 所指向的字符串的前 **n** 个字节中搜索第一次出现字符 **c**（一个无符号字符）的位置。

声明

下面是 memchr() 函数的声明。

```
void *memchr(const void *str, int c, size_t n)
```

参数

- **str** -- 指向要执行搜索的内存块。
- **c** -- 以 int 形式传递的值，但是函数在每次字节搜索时是使用该值的无符号字符形式。
- **n** -- 要被分析的字节数。

返回值

该函数返回一个指向匹配字节的指针，如果在给定的内存区域未出现字符，则返回 NULL。

实例

下面的实例演示了 memchr() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   const char str[] = "http://www.w3cschool.cc";
   const char ch = '.';
   char *ret;

   ret = memchr(str, ch, strlen(str));

   printf("|%c| 之后的字符串是 - |%s|\n", ch, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
|.| 之后的字符串是 - |.w3cschool.cc|
```

C 库函数 - memcmp()

描述

C 库函数 **int memcmp(const void *str1, const void *str2, size_t n))** 把存储区 **str1** 和存储区 **str2** 的前 **n** 个字节进行比较。

声明

下面是 memcmp() 函数的声明。

```
int memcmp(const void *str1, const void *str2, size_t n)
```

参数

- **str1** -- 指向内存块的指针。
- **str2** -- 指向内存块的指针。
- **n** -- 要被比较的字节数。

返回值

- 如果返回值 < 0，则表示 str1 小于 str2。
- 如果返回值 > 0，则表示 str2 小于 str1。
- 如果返回值 = 0，则表示 str1 等于 str2。

实例

下面的实例演示了 memcmp() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[15];
   char str2[15];
   int ret;

   memcpy(str1, "abcdef", 6);
   memcpy(str2, "ABCDEF", 6);

   ret = memcmp(str1, str2, 5);

   if(ret > 0)
   {
      printf("str2 小于 str1");
   }
   else if(ret > 0)
   {
      printf("str1 小于 str2");
   }
   else
   {
      printf("str1 等于 str2");
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
str2 小于 str1
```

C 库函数 - memcpy()

描述

C 库函数 **void *memcpy(void *str1, const void *str2, size_t n)** 从存储区 **str2** 复制 **n** 个字符到存储区 **str1**。

声明

下面是 memcpy() 函数的声明。

```
void *memcpy(void *str1, const void *str2, size_t n)
```

参数

- **str1** -- 指向用于存储复制内容的目标数组，类型强制转换为 void* 指针。
- **str2** -- 指向要复制的数据源，类型强制转换为 void* 指针。
- **n** -- 要被复制的字节数。

返回值

该函数返回一个指向目标存储区 str1 的指针。

实例

下面的实例演示了 memcpy() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   const char src[50] = "http://www.w3cschool.cc";
   char dest[50];

   printf("Before memcpy dest = %s\n", dest);
   memcpy(dest, src, strlen(src)+1);
   printf("After memcpy dest = %s\n", dest);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Before memcpy dest =
After memcpy dest = http://www.w3cschool.cc
```

C 库函数 - memmove()

描述

C 库函数 **void *memmove(void* str1, const void *str2, size_t n)** 从 **str2** 复制 **n** 个字符到 **str1**，但是在重叠内存块这方面，memmove() 是比 memcpy() 更安全的方法。如果目标区域和源区域有重叠的话，memmove() 能够保证源串在被覆盖之前将重叠区域的字节拷贝到目标区域中，复制后源区域的内容会被更改。如果目标区域与源区域没有重叠，则和 memcpy() 函数功能相同。

声明

下面是 memmove() 函数的声明。

```
void *memmove(void *str1, const void *str2, size_t n)
```

参数

- **str1** -- 指向用于存储复制内容的目标数组，类型强制转换为 void* 指针。
- **str2** -- 指向要复制的数据源，类型强制转换为 void* 指针。
- **n** -- 要被复制的字节数。

返回值

该函数返回一个指向目标存储区 str1 的指针。

实例

下面的实例演示了 memmove() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   const char dest[] = "oldstring";
   const char src[]  = "newstring";

   printf("Before memmove dest = %s, src = %s\n", dest, src);
   memmove(dest, src, 9);
   printf("After memmove dest = %s, src = %s\n", dest, src);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Before memmove dest = oldstring, src = newstring
After memmove dest = newstring, src = newstring
```

C 库函数 - memset()

描述

C 库函数 **void *memset(void *str, int c, size_t n)** 复制字符 **c**（一个无符号字符）到参数 **str** 所指向的字符串的前 **n** 个字符。

声明

下面是 memset() 函数的声明。

```
void *memset(void *str, int c, size_t n)
```

参数

- **str** -- 指向要填充的内存块。
- **c** -- 要被设置的值。该值以 int 形式传递，但是函数在填充内存块时是使用该值的无符号字符形式。
- **n** -- 要被设置为该值的字节数。

返回值

该值返回一个指向存储区 str 的指针。

实例

下面的实例演示了 memset() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str[50];

   strcpy(str,"This is string.h library function");
   puts(str);

   memset(str,'$',7);
   puts(str);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
This is string.h library function
$$$$$$$ string.h library function
```

C 库函数 - strcat()

描述

C 库函数 **char *strcat(char* dest, const char *src)** 把 **src** 所指向的字符串追加到 **dest** 所指向的字符串的结尾。

声明

下面是 strcat() 函数的声明。

```
char *strcat(char *dest, const char *src)
```

参数

- **dest** -- 指向目标数组，该数组包含了一个 C 字符串，且足够容纳追加后的字符串。
- **src** -- 指向要追加的字符串，该字符串不会覆盖目标字符串。

返回值

该函数返回一个指向最终的目标字符串 dest 的指针。

实例

下面的实例演示了 strcat() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char src[50], dest[50];

   strcpy(src,  "This is source");
   strcpy(dest, "This is destination");

   strcat(dest, src);

   printf("最终的目标字符串： |%s|", dest);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
最终的目标字符串： |This is destinationThis is source|
```

C 库函数 - strncat()

描述

C 库函数 **char *strncat(char *dest, const char *src, size_t n)** 把 **src** 所指向的字符串追加到 **dest** 所指向的字符串的结尾，直到 **n** 字符长度为止。

声明

下面是 strncat() 函数的声明。

```
char *strncat(char *dest, const char *src, size_t n)
```

参数

- **dest** -- 指向目标数组，该数组包含了一个 C 字符串，且足够容纳追加后的字符串，包括额外的空字符。
- **src** -- 要追加的字符串。
- **n** -- 要追加的最大字符数。

返回值

该函数返回一个指向最终的目标字符串 dest 的指针。

实例

下面的实例演示了 strncat() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char src[50], dest[50];

   strcpy(src,  "This is source");
   strcpy(dest, "This is destination");

   strncat(dest, src, 15);

   printf("最终的目标字符串： |%s|", dest);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
最终的目标字符串： |This is destinationThis is source|
```

C 库函数 - strchr()

描述

C 库函数 **char *strchr(const char* str, int c)** 在参数 **str** 所指向的字符串中搜索第一次出现字符 **c**（一个无符号字符）的位置。

声明

下面是 strchr() 函数的声明。

```
char *strchr(const char *str, int c)
```

参数

- **str** -- 要被检索的 C 字符串。
- **c** -- 在 str 中要搜索的字符。

返回值

该函数返回在字符串 str 中第一次出现字符 c 的位置，如果未找到该字符则返回 NULL。

实例

下面的实例演示了 strchr() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   const char str[] = "http://www.w3cschool.cc";
   const char ch = '.';
   char *ret;

   ret = strchr(str, ch);

   printf("|%c| 之后的字符串是 - |%s|\n", ch, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
|.| 之后的字符串是 - |.w3cschool.cc|
```

C 库函数 - strcmp()

描述

C 库函数 **int strcmp(const char** ***str1, const char*** **str2)** 把 **str1** 所指向的字符串和 **str2** 所指向的字符串进行比较。

声明

下面是 strcmp() 函数的声明。

```
int strcmp(const char *str1, const char *str2)
```

参数

- **str1** -- 要进行比较的第一个字符串。
- **str2** -- 要进行比较的第二个字符串。

返回值

该函数返回值如下：

- 如果返回值 < 0，则表示 str1 小于 str2。
- 如果返回值 > 0，则表示 str2 小于 str1。
- 如果返回值 = 0，则表示 str1 等于 str2。

实例

下面的实例演示了 strncmp() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[15];
   char str2[15];
   int ret;

   strcpy(str1, "abcdef");
   strcpy(str2, "ABCDEF");

   ret = strcmp(str1, str2);

   if(ret < 0)
   {
      printf("str1 小于 str2");
   }
   else if(ret > 0)
   {
      printf("str2 小于 str1");
   }
   else
   {
      printf("str1 等于 str2");
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
str2 小于 str1
```

C 库函数 - strncmp()

描述

C 库函数 **int strncmp(const char** ***str1, const char*** **str2, size_t n)** 把 **str1** 和 **str2** 进行比较，最多比较前 **n** 个字节。

声明

下面是 strncmp() 函数的声明。

```
int strncmp(const char *str1, const char *str2, size_t n)
```

参数

- **str1** -- 要进行比较的第一个字符串。
- **str2** -- 要进行比较的第二个字符串。
- **n** -- 要比较的最大字符数。

返回值

该函数返回值如下：

- 如果返回值 < 0，则表示 str1 小于 str2。
- 如果返回值 > 0，则表示 str2 小于 str1。
- 如果返回值 = 0，则表示 str1 等于 str2。

实例

下面的实例演示了 strncmp() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[15];
   char str2[15];
   int ret;

   strcpy(str1, "abcdef");
   strcpy(str2, "ABCDEF");

   ret = strncmp(str1, str2, 4);

   if(ret < 0)
   {
      printf("str1 小于 str2");
   }
   else if(ret > 0)
   {
      printf("str2 小于 str1");
   }
   else
   {
      printf("str1 等于 str2");
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
str2 小于 str1
```

C 库函数 - strcoll()

描述

C 库函数 **int strcoll(const char** ***str1, const char*** **str2)** 把 **str1** 和 **str2** 进行比较，结果取决于 LC_COLLATE 的位置设置。

声明

下面是 strcoll() 函数的声明。

```
int strcoll(const char *str1, const char *str2)
```

参数

- **str1** -- 要进行比较的第一个字符串。
- **str2** -- 要进行比较的第二个字符串。

返回值

该函数返回值如下：

- 如果返回值 < 0，则表示 str1 小于 str2。
- 如果返回值 > 0，则表示 str2 小于 str1。
- 如果返回值 = 0，则表示 str1 等于 str2。

实例

下面的实例演示了 strcoll() 函数的用法。.

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[15];
   char str2[15];
   int ret;

   strcpy(str1, "abc");
   strcpy(str2, "ABC");

   ret = strcoll(str1, str2);

   if(ret > 0)
   {
      printf("str1 小于 str2");
   }
   else if(ret < 0)
   {
      printf("str2 小于 str1");
   }
   else
   {
      printf("str1 等于 str2");
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
str1 小于 str2
```

C 库函数 - strcpy()

描述

C 库函数 **char *strcpy(char *dest, const char *src)** 把 **src** 所指向的字符串复制到 **dest**。

声明

下面是 strcpy() 函数的声明。

```
char *strcpy(char *dest, const char *src)
```

参数

- **dest** -- 指向用于存储复制内容的目标数组。
- **src** -- 要复制的字符串。

返回值

该函数返回一个指向最终的目标字符串 dest 的指针。

实例

下面的实例演示了 strcpy() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main()
{
   char src[40];
   char dest[100];

   memset(dest, '\0', sizeof(dest));
   strcpy(src, "This is w3cschool.cc");
   strcpy(dest, src);

   printf("最终的目标字符串： %s\n", dest);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
最终的目标字符串： This is w3cschool.cc
```

C 库函数 - strncpy()

描述

C 库函数 **char *strncpy(char *dest, const char *src, size_t n)** 把 **src** 所指向的字符串复制到 **dest**，最多复制 **n** 个字符。当 src 的长度小于 n 时，dest 的剩余部分将用空字节填充。

声明

下面是 strncpy() 函数的声明。

```
char *strncpy(char *dest, const char *src, size_t n)
```

参数

- **dest** -- 指向用于存储复制内容的目标数组。
- **src** -- 要复制的字符串。
- **n** -- 要从源中复制的字符数。

返回值

该函数返回最终复制的字符串。

实例

下面的实例演示了 strncpy() 函数的用法。在这里，我们使用函数 memset() 来清除内存位置。

```
#include <stdio.h>
#include <string.h>

int main()
{
   char src[40];
   char dest[12];

   memset(dest, '\0', sizeof(dest));
   strcpy(src, "This is w3cschool.cc");
   strncpy(dest, src, 10);

   printf("最终的目标字符串： %s\n", dest);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
最终的目标字符串： This is w3
```

C 库函数 - strcspn()

描述

C 库函数 **size_t strcspn(const char *str1, const char* str2)** 检索字符串 **str1** 开头连续有几个字符都不含字符串 **str2** 中的字符。

声明

下面是 strcspn() 函数的声明。

```
size_t strcspn(const char *str1, const char *str2)
```

参数

- **str1** -- 要被检索的 C 字符串。
- **str2** -- 该字符串包含了要在 str1 中进行匹配的字符列表。

返回值

该函数返回 str1 开头连续都不含字符串 str2 中字符的字符数。

实例

下面的实例演示了 strcspn() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   int len;
   const char str1[] = "ABCDEF4960910";
   const char str2[] = "013";

   len = strcspn(str1, str2);

   printf("第一个匹配的字符是在 %d\n", len + 1);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
第一个匹配的字符是在 10
```

C 库函数 - strerror()

描述

C 库函数 **char *strerror(int errnum)** 从内部数组中搜索错误号 **errnum**，并返回一个指向错误消息字符串的指针。**strerror** 生成的错误字符串取决于开发平台和编译器。

声明

下面是 strerror() 函数的声明。

```
char *strerror(int errnum)
```

参数

- **errnum** -- 错误号，通常是 **errno**。

返回值

该函数返回一个指向错误字符串的指针，该错误字符串描述了错误 errnum。

实例

下面的实例演示了 strerror() 函数的用法。

```
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main ()
{
   FILE *fp;

   fp = fopen("file.txt","r");
   if( fp == NULL )
   {
      printf("Error: %s\n", strerror(errno));
   }

  return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果，因为我们尝试打开一个不存在的文件：

```
Error: No such file or directory
```

C 库函数 - strlen()

描述

C 库函数 **size_t strlen(const char *str)** 计算字符串 **str** 的长度，直到空结束字符，但不包括空结束字符。

声明

下面是 strlen() 函数的声明。

```
size_t strlen(const char *str)
```

参数

- **str** -- 要计算长度的字符串。

返回值

该函数返回字符串的长度。

实例

下面的实例演示了 strlen() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str[50];
   int len;

   strcpy(str, "This is w3cschool.cc");

   len = strlen(str);
   printf("|%s| 的长度是 |%d|\n", str, len);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
|This is w3cschool.cc| 的长度是 |20|
```

C 库函数 - strpbrk()

描述

C 库函数 **char *strpbrk(const char* str1, const char *str2)** 检索字符串 **str1** 中第一个匹配字符串 **str2** 中字符的字符，不包含空结束字符。也就是说，依次检验字符串 str1 中的字符，当被检验字符在字符串 str2 中也包含时，则停止检验，并返回该字符位置。

声明

下面是 strpbrk() 函数的声明。

```
char *strpbrk(const char *str1, const char *str2)
```

参数

- **str1** -- 要被检索的 C 字符串。
- **str2** -- 该字符串包含了要在 str1 中进行匹配的字符列表。

返回值

该函数返回 str1 中第一个匹配字符串 str2 中字符的字符数，如果未找到字符则返回 NULL。

实例

下面的实例演示了 strpbrk() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   const char str1[] = "abcde2fghi3jk4l";
   const char str2[] = "34";
   char *ret;

   ret = strpbrk(str1, str2);
   if(ret)
   {
      printf("第一个匹配的字符是： %c\n", *ret);
   }
   else
   {
      printf("未找到字符");
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
第一个匹配的字符是： 3
```

C 库函数 - strrchr()

描述

C 库函数 **char *strrchr(const char* str, int c)** 在参数 **str** 所指向的字符串中搜索最后一次出现字符 **c**（一个无符号字符）的位置。

声明

下面是 strrchr() 函数的声明。

```
char *strrchr(const char *str, int c)
```

参数

- **str** -- C 字符串。
- **c** -- 要搜索的字符。以 int 形式传递，但是最终会转换回 char 形式。

返回值

该函数返回 str 中最后一次出现字符 c 的位置。如果未找到该值，则函数返回一个空指针。

实例

下面的实例演示了 strrchr() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   int len;
   const char str[] = "http://www.w3cschool.cc";
   const char ch = '.';
   char *ret;

   ret = strrchr(str, ch);

   printf("|%c| 之后的字符串是 - |%s|\n", ch, ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
|.| 之后的字符串是 - |.cc|
```

C 库函数 - strspn()

描述

C 库函数 **size_t strspn(const char *str1, const char str2)* 检索字符串 **str1** 中第一个不在字符串 **str2** 中出现的字符下标。

声明

下面是 strspn() 函数的声明。

```
size_t strspn(const char *str1, const char *str2)
```

参数

- **str1** -- 要被检索的 C 字符串。
- **str2** -- 该字符串包含了要在 str1 中进行匹配的字符列表。

返回值

该函数返回 str1 中第一个不在字符串 str2 中出现的字符下标。

实例

下面的实例演示了 strspn() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main ()
{
   int len;
   const char str1[] = "ABCDEFG019874";
   const char str2[] = "ABCD";

   len = strspn(str1, str2);

   printf("初始段匹配长度 %d\n", len );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
初始段匹配长度 4
```

C 库函数 - strstr()

描述

C 库函数 **char *strstr(const char* haystack, const char *needle)** 在字符串 **haystack** 中查找第一次出现字符串 **needle** 的位置，不包含终止符 '\0'。

声明

下面是 strstr() 函数的声明。

```
char *strstr(const char *haystack, const char *needle)
```

参数

- **haystack** -- 要被检索的 C 字符串。
- **needle** -- 在 haystack 字符串内要搜索的小字符串。

返回值

该函数返回在 haystack 中第一次出现 needle 字符串的位置，如果未找到则返回 null。

实例

下面的实例演示了 strstr() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main()
{
   const char haystack[20] = "W3CSchool";
   const char needle[10] = "School";
   char *ret;

   ret = strstr(haystack, needle);

   printf("子字符串是： %s\n", ret);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
子字符串是： School
```

C 库函数 - strtok()

描述

C 库函数 **char** ***strtok(char*** **str, const char *delim)** 分解字符串 **str** 为一组字符串，**delim** 为分隔符。

声明

下面是 strtok() 函数的声明。

```
char *strtok(char *str, const char *delim)
```

参数

- **str** -- 要被分解成一组小字符串的字符串。
- **delim** -- 包含分隔符的 C 字符串。

返回值

该函数返回被分解的最后一个子字符串，如果没有可检索的字符串，则返回一个空指针。

实例

下面的实例演示了 strtok() 函数的用法。

```
#include <string.h>
#include <stdio.h>

int main()
{
   const char str[80] = "This is - www.w3cschool.cc - website";
   const char s[2] = "-";
   char *token;

   /* 获取第一个子字符串 */
   token = strtok(str, s);

   /* 继续获取其他的子字符串 */
   while( token != NULL )
   {
      printf( " %s\n", token );

      token = strtok(NULL, s);
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
This is
www.w3cschool.cc
website
```

C 库函数 - strxfrm()

描述

C 库函数 **size_t strxfrm(char *dest, const char* src, size_t n)** 根据程序当前的区域选项中的 LC_COLLATE 来转换字符串 **src** 的前 **n** 个字符，并把它们放置在字符串 **dest** 中。

声明

下面是 strxfrm() 函数的声明。

```
size_t strxfrm(char *dest, const char *src, size_t n)
```

参数

- **dest** -- 指向存储内容的目标数组的指针，如果参数 n 为 0，则它是一个空指针。
- **src** -- 要被转换为当前区域设置的 C 字符串。
- **n** -- 被复制到 str1 的最大字符数。

返回值

该函数返回被转换字符串的长度，不包括空结束字符。

实例

下面的实例演示了 strxfrm() 函数的用法。

```
#include <stdio.h>
#include <string.h>

int main()
{
   char dest[20];
   char src[20];
   int len;

   strcpy(src, "W3C School");
   len = strxfrm(dest, src, 20);

   printf("字符串 |%s| 的长度是： |%d|", dest, len);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
字符串 |W3C School| 的长度是： |10|
```

C 标准库 - <time.h>

简介

time.h 头文件定义了四个变量类型、两个宏和各种操作日期和时间的函数。

库变量

下面是头文件 time.h 中定义的变量类型：

变量	描述
size_t	是无符号整数类型，它是 **sizeof** 关键字的结果。
clock_t	这是一个适合存储处理器时间的类型。
time_t is	这是一个适合存储日历时间类型。
struct tm	这是一个用来保存时间和日期的结构。

tm 结构的定义如下：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59        */
   int tm_min;         /* 分，范围从 0 到 59        */
   int tm_hour;        /* 小时，范围从 0 到 23        */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31    */
   int tm_mon;         /* 月，范围从 0 到 11        */
   int tm_year;        /* 自 1900 年起的年数        */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6    */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365    */
   int tm_isdst;       /* 夏令时               */
};
```

库宏

下面是头文件 time.h 中定义的宏：

宏	描述
NULL	这个宏是一个空指针常量的值。
CLOCKS_PER_SEC	这个宏表示每秒的处理器时钟个数。

库函数

下面是头文件 time.h 中定义的函数：

函数	描述
char *asctime(const struct tm *timeptr)	返回一个指向字符串的指针，它代表了结构 timeptr 的日期和时间。
clock_t clock(void)	返回程序执行起（一般为程序的开头），处理器时钟所使用的时间。
char *ctime(const time_t *timer)	返回一个表示当地时间的字符串，当地时间是基于参数 timer。
double difftime(time_t time1, time_t time2)	返回 time1 和 time2 之间相差的秒数 (time1-time2)。
struct tm *gmtime(const time_t *timer)	timer 的值被分解为tm 结构，并用协调世界时（UTC）也被称为格林尼治标准时间（GMT）表示。
struct tm *localtime(const time_t *timer)	timer 的值被分解为tm 结构，并用本地时区表示。
time_t mktime(struct tm *timeptr)	把 timeptr 所指向的结构转换为一个依据本地时区的 time_t 值。
size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr)	根据 format 中定义的格式化规则，格式化结构 timeptr 表示的时间，并把它存储在 str 中。
time_t time(time_t *timer)	计算当前日历时间，并把它编码成 time_t 格式。

C 库函数 - asctime()

描述

C 库函数 **char *asctime(const struct tm* timeptr)** 返回一个指向字符串的指针，它代表了结构 **struct timeptr** 的日期和时间。

声明

下面是 asctime() 函数的声明。

```
char *asctime(const struct tm *timeptr)
```

参数

timeptr 是指向 tm 结构的指针，包含了分解为如下各部分的日历时间：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59                */
   int tm_min;         /* 分，范围从 0 到 59                */
   int tm_hour;        /* 小时，范围从 0 到 23                */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31                    */
   int tm_mon;         /* 月份，范围从 0 到 11                */
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6                */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365                    */
   int tm_isdst;       /* 夏令时                        */
};
```

返回值

该函数返回一个 C 字符串，包含了可读格式的日期和时间信息 **Www Mmm dd hh:mm:ss yyyy**，其中，*Www* 表示星期几，*Mmm* 是以字母表示的月份，*dd* 表示一月中的第几天，*hh:mm:ss* 表示时间，*yyyy* 表示年份。

实例

下面的实例演示了 asctime() 函数的用法。

```
#include <stdio.h>
#include <string.h>
#include <time.h>

int main()
{
   struct tm t;

   t.tm_sec    = 10;
   t.tm_min    = 10;
   t.tm_hour   = 6;
   t.tm_mday   = 25;
   t.tm_mon    = 2;
   t.tm_year   = 89;
   t.tm_wday   = 6;

   puts(asctime(&t));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Sat Mar 25 06:10:10 1989
```

C 库函数 - clock()

描述

C 库函数 **clock_t clock(void)** 返回程序执行起（一般为程序的开头），处理器时钟所使用的时间。为了获取 CPU 所使用的秒数，您需要除以 CLOCKS_PER_SEC。

在 32 位系统中，CLOCKS_PER_SEC 等于 1000000，该函数大约每 72 分钟会返回相同的值。

声明

下面是 clock() 函数的声明。

```
clock_t clock(void)
```

参数

- **NA**

返回值

该函数返回自程序启动起，处理器时钟所使用的时间。如果失败，则返回 -1 值。

实例

下面的实例演示了 clock() 函数的用法。

```
#include <time.h>
#include <stdio.h>

int main()
{
   clock_t start_t, end_t, total_t;
   int i;

   start_t = clock();
   printf("程序启动，start_t = %ld\n", start_t);

   printf("开始一个大循环，start_t = %ld\n", start_t);
   for(i=0; i< 10000000; i++)
   {
   }
   end_t = clock();
   printf("大循环结束，end_t = %ld\n", end_t);

   total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
   printf("CPU 占用的总时间：%f\n", total_t  );
   printf("程序退出...\n");

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
程序启动，start_t = 0
开始一个大循环，start_t = 0
大循环结束，end_t = 20000
CPU 占用的总时间：0.000000
程序退出...
```

C 库函数 - ctime()

描述

C 库函数 **char *ctime(const time_t* timer)** 返回一个表示当地时间的字符串，当地时间是基于参数 **timer**。

返回的字符串格式如下： **Www Mmm dd hh:mm:ss yyyy** 其中，*Www* 表示星期几，*Mmm* 是以字母表示的月份，*dd* 表示一月中的第几天，*hh:mm:ss* 表示时间，*yyyy* 表示年份。

声明

下面是 ctime() 函数的声明。

```
char *ctime(const time_t *timer)
```

参数

- **timer** -- 这是指向 time_t 对象的指针，该对象包含了一个日历时间。

返回值

该函数返回一个 C 字符串，该字符串包含了可读格式的日期和时间信息。

实例

下面的实例演示了 ctime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t curtime;

   time(&curtime);

   printf("当前时间 = %s", ctime(&curtime));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
当前时间 = Mon Aug 13 08:23:14 2012
```

C 库函数 - difftime()

描述

C 库函数 **double difftime(time_t time1, time_t time2)** 返回 **time1** 和 **time2** 之间相差的秒数 **(time1 - time2)**。这两个时间是在日历时间中指定的，表示了自纪元 Epoch（协调世界时 UTC：1970-01-01 00:00:00）起经过的时间。

声明

下面是 difftime() 函数的声明。

```
double difftime(time_t time1, time_t time2)
```

参数

- **time1** -- 这是表示结束时间的 time_t 对象。
- **time2** -- 这是表示开始时间的 time_t 对象。

返回值

该函数返回以双精度浮点型 double 值表示的两个时间之间相差的秒数 (time2 - time1)。

实例

下面的实例演示了 difftime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t start_t, end_t;
   double diff_t;

   printf("程序启动...\n");
   time(&start_t);

   printf("休眠 5 秒...\n");
   sleep(5);

   time(&end_t);
   diff_t = difftime(end_t, start_t);

   printf("执行时间 = %f\n", diff_t);
   printf("程序退出...\n");

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
程序启动...
休眠 5 秒...
执行时间 = 5.000000
程序退出...
```

C 库函数 - gmtime()

描述

C 库函数 **struct tm *gmtime(const time_t* timer)** 使用 **timer** 的值来填充 **tm** 结构，并用协调世界时（UTC）也被称为格林尼治标准时间（GMT）表示。

声明

下面是 gmtime() 函数的声明。

```
struct tm *gmtime(const time_t *timer)
```

参数

- **timeptr** -- 这是指向表示日历时间的 time_t 值的指针。

返回值

该函数返回指向 tm 结构的指针，该结构带有被填充的时间信息。下面是 timeptr 结构的细节：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59                */
   int tm_min;         /* 分，范围从 0 到 59                */
   int tm_hour;        /* 小时，范围从 0 到 23                */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31                    */
   int tm_mon;         /* 月份，范围从 0 到 11                */
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6                */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365                    */
   int tm_isdst;       /* 夏令时                        */
};
```

实例

下面的实例演示了 gmtime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

#define BST (+1)
#define CCT (+8)

int main ()
{

   time_t rawtime;
   struct tm *info;

   time(&rawtime);
   /* 获取 GMT 时间 */
   info = gmtime(&rawtime );

   printf("当前的世界时钟：\n");
   printf("伦敦：%2d:%02d\n", (info->tm_hour+BST)%24, info->tm_min);
   printf("中国：%2d:%02d\n", (info->tm_hour+CCT)%24, info->tm_min);

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
当前的世界时钟：
伦敦：14:10
中国：21:10
```

C 库函数 - localtime()

描述

C 库函数 **struct tm *localtime(const time_t *timer)** 使用 timer 的值来填充 **tm** 结构。**timer** 的值被分解为 **tm** 结构，并用本地时区表示。

声明

下面是 localtime() 函数的声明。

```
struct tm *localtime(const time_t *timer)
```

参数

- **timer** -- 这是指向表示日历时间的 time_t 值的指针。

返回值

该函数返回指向 **tm** 结构的指针，该结构带有被填充的时间信息。下面是 tm 结构的细节：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59                */
   int tm_min;         /* 分，范围从 0 到 59                */
   int tm_hour;        /* 小时，范围从 0 到 23                */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31                    */
   int tm_mon;         /* 月份，范围从 0 到 11                */
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6                */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365                    */
   int tm_isdst;       /* 夏令时                        */
};
```

实例

下面的实例演示了 localtime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t rawtime;
   struct tm *info;
   char buffer[80];

   time( &rawtime );

   info = localtime( &rawtime );
   printf("当前的本地时间和日期：%s", asctime(info));

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
当前的本地时间和日期：Thu Aug 23 09:12:05 2012
```

C 库函数 - mktime()

描述

C 库函数 **time_t mktime(struct tm *timeptr)** 把 **timeptr** 所指向的结构转换为一个依据本地时区的 time_t 值。

声明

下面是 mktime() 函数的声明。

```
time_t mktime(struct tm *timeptr)
```

参数

- **timeptr** -- 这是指向表示日历时间的 time_t 值的指针，该日历时间被分解为以下各部分。下面是 timeptr 结构的细节：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59                */
   int tm_min;         /* 分，范围从 0 到 59                */
   int tm_hour;        /* 小时，范围从 0 到 23                */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31                */
   int tm_mon;         /* 月份，范围从 0 到 11                */
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6                */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365                */
   int tm_isdst;       /* 夏令时                        */
};
```

返回值

该函数返回一个 time_t 值，该值对应于以参数传递的日历时间。如果发生错误，则返回 -1 值。

实例

下面的实例演示了 mktime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   int ret;
   struct tm info;
   char buffer[80];

   info.tm_year = 2001 - 1900;
   info.tm_mon = 7 - 1;
   info.tm_mday = 4;
   info.tm_hour = 0;
   info.tm_min = 0;
   info.tm_sec = 1;
   info.tm_isdst = -1;

   ret = mktime(&info);
   if( ret == -1 )
   {
       printf("错误：不能使用 mktime 转换时间。\n");
   }
   else
   {
      strftime(buffer, sizeof(buffer), "%c", &info );
      print(buffer);
   }

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
Wed Jul 4 00:00:01 2001
```

C 库函数 - strftime()

描述

C 库函数 **size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr)** 根据 **format** 中定义的格式化规则，格式化结构 **timeptr** 表示的时间，并把它存储在 **str** 中。

声明

下面是 strftime() 函数的声明。

```
size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr)
```

参数

- **str** -- 这是指向目标数组的指针，用来复制产生的 C 字符串。
- **maxsize** -- 这是被复制到 str 的最大字符数。
- **format** -- 这是 C 字符串，包含了普通字符和特殊格式说明符的任何组合。这些格式说明符由函数替换为表示 tm 中所指定时间的相对应值。格式说明符是：

说明符	替换为	实例
%a	缩写的星期几名称	Sun
%A	完整的星期几名称	Sunday
%b	缩写的月份名称	Mar
%B	完整的月份名称	March
%c	日期和时间表示法	Sun Aug 19 02:56:02 2012
%d	一月中的第几天（01-31）	19
%H	24 小时格式的小时（00-23）	14
%I	12 小时格式的小时（01-12）	05
%j	一年中的第几天（001-366）	231
%m	十进制数表示的月份（01-12）	08
%M	分（00-59）	55
%p	AM 或 PM 名称	PM
%S	秒（00-61）	02
%U	一年中的第几周，以第一个星期日作为第一周的第一天（00-53）	33
%w	十进制数表示的星期几，星期日表示为 0（0-6）	4
%W	一年中的第几周，以第一个星期一作为第一周的第一天（00-53）	34
%x	日期表示法	08/19/12
%X	时间表示法	02:50:06
%y	年份，最后两个数字（00-99）	01
%Y	年份	2012
%Z	时区的名称或缩写	CDT
%%	一个 % 符号	%

- **timeptr** -- 这是指向 tm 结构的指针，该结构包含了一个呗分解为以下各部分的日历时间：

```
struct tm {
   int tm_sec;         /* 秒，范围从 0 到 59                */
   int tm_min;         /* 分，范围从 0 到 59                */
   int tm_hour;        /* 小时，范围从 0 到 23                */
   int tm_mday;        /* 一月中的第几天，范围从 1 到 31                    */
   int tm_mon;         /* 月份，范围从 0 到 11                */
   int tm_year;        /* 自 1900 起的年数                */
   int tm_wday;        /* 一周中的第几天，范围从 0 到 6                */
   int tm_yday;        /* 一年中的第几天，范围从 0 到 365                    */
   int tm_isdst;       /* 夏令时                        */
};
```

返回值

如果产生的 C 字符串小于 size 个字符（包括空结束字符），则会返回复制到 str 中的字符总数（不包括空结束字符），否则返回零。

实例

下面的实例演示了 strftime() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
   time_t rawtime;
   struct tm *info;
   char buffer[80];

   time( &rawtime );

   info = localtime( &rawtime );

   strftime(buffer,80,"%x - %I:%M%p", info);
   printf("格式化的日期 & 时间 : |%s|\n", buffer );

   return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
格式化的日期 & 时间 : |08/23/12 - 12:40AM|
```

C 库函数 - time()

描述

C 库函数 **time_t time(time_t *seconds)** 返回自纪元 Epoch（1970-01-01 00:00:00 UTC）起经过的时间，以秒为单位。如果 **seconds** 不为空，则返回值也存储在变量 **seconds** 中。

声明

下面是 time() 函数的声明。

```
time_t time(time_t *t)
```

参数

- **seconds** -- 这是指向类型为 time_t 的对象的指针，用来存储 seconds 的值。

返回值

以 time_t 对象返回当前日历时间。

实例

下面的实例演示了 time() 函数的用法。

```
#include <stdio.h>
#include <time.h>

int main ()
{
  time_t seconds;

  seconds = time(NULL);
  printf("自 1970-01-01 起的小时数 = %ld\n", seconds/3600);

  return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果：

```
自 1970-01-01 起的小时数 = 373711
```

免责声明

W3School提供的内容仅用于培训。我们不保证内容的正确性。通过使用本站内容随之而来的风险与本站无关。W3School简体中文版的所有内容仅供测试，对任何法律问题及风险不承担任何责任。